AD A260 693

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

| | | |
|---|---|---|
| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE 10/19/92 | 3. REPORT TYPE AND DATES COVERED |

| | |
|---|---|
| 4. TITLE AND SUBTITLE Very High-Speed Arithmetic Processors | 5. FUNDING NUMBERS |
| 6. AUTHOR(S) Dr. F.J. Taylor | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Florida, 219 Grinter Hall, Gainesville, FL 32611 | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Research Office, P. O. Box 12211, Research Triangle Park, NC 27709-2211 | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER |

11. SUPPLEMENTARY NOTES
The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

| | |
|---|---|
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited. | 12b. DISTRIBUTION CODE |

13. ABSTRACT *(Maximum 200 words)*
Our VHSAP ARO study has lead to a number of accomplishments including an original VLSI processor and a system with a high residue number system (RNS) content called the Gauss Machine. The Gauss machine is a SIMD systolic array architecture which takes advantage of the Galois-enhanced quadratic residue number system (GEQRNS) to form reduced complexity arithmetic elements. The Gauss machine is targeted at front-end signal and image processing applications. With a 2x2 array of GEQRNS multiplier-accumulators operating at 10 MHz, the Gauss machine can achieve a peak equivalent throughput of 320 million operations per second when performing complex arithmetic. The Gauss machine is designed for a broader, more general class of problems other than RNS based systems which have been constructed: the Gauss machine may be used to accelerate computations which involve or may be expressed as matrix-matrix (level 3), matrix-vector (level 2), or vector-vector (level 1) operations. This paper describes the implementation of the Gauss machine and how it may be used to accelerate signal processing operations.

| | |
|---|---|
| 14. SUBJECT TERMS DODFARSUP52.235-7005 FAR52.216.7 | 15. NUMBER OF PAGES 165 |
| | 16. PRICE CODE |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

# VERY HIGH-SPEED ARITHMETIC PROCESSORS
# ( THE GAUSS MACHINE)

FINAL REPORT

DR. FRED J. TAYLOR

10/29/92

DAAL03-89-K-0147

A REPORT PRESENTED TO THE ARMY RESEARCH OFFICE.
PREPARED BY THE HIGH-SPEED DIGITAL ARCHITECTURE LABORATORY.

UNIVERSITY OF FLORIDA

1992

## ABSTRACT

## VERY HIGH-SPEED ARITHMETIC PROCESSORS

Our VHSAP ARO study has lead to a number of accomplishments including an original VLSI processor and a system with a high residue number system (RNS) content called the Gauss Machine. The Gauss machine is a SIMD systolic array architecture which takes advantage of the Galois-enhanced quadratic residue number system (GEQRNS) to form reduced complexity arithmetic elements. The Gauss machine is targeted at front-end signal and image processing applications. With a $2 \times 2$ array of GEQRNS multiplier-accumulators operating at 10 MHz, the Gauss machine can achieve a peak equivalent throughput of 320 million operations per second when performing complex arithmetic. The Gauss machine is designed for a broader, more general class of problems than other RNS based systems which have been constructed: the Gauss machine may be used to accelerate computations which involve or may be expressed as matrix-matrix (level 3), matrix-vector (level 2), or vector-vector (level 1) operations. This paper describes the implementation of the Gauss machine and how it may be used to accelerate signal processing operations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# Part I

# Theory

Chapter 1

# BASIS OF RESIDUE NUMBER SYSTEM

## 1.1 The Chinese Remainder Theorem

There are two large penalties in performing arithmetic in the two's complement system: the carry must propagate across the entire word for addition operations, and the size of the multiplier grows as the square of the width of the word. The Chinese Remainder Theorem (CRT) [1, 2] suggests a means of eliminating the carry propagation problem and of producing a multiplier that grows linearly with the width of the word. The CRT is presented below.

**Theorem 1 (The Chinese Remainder Theorem)** *Let* $M = \prod_{i=1}^{L} p_i$, *where for* $i, j \in \{1, 2, 3, \ldots, L\}$, $\gcd(p_i, p_j) = 1$ *for all* $i \neq j$, *and each* $p_i \in \mathbf{Z}^+$. *Then there exists an isomorphism* $\phi$: $\mathbf{Z}_M \leftrightarrow \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L}$ *described by the following.*

*Let* $m_i = M/p_i$, *and* $m_i m_i^{-1} \equiv 1 \pmod{p_i}$ *for all* $i \in \{1, 2, 3, \ldots, L\}$. *If* $X \in \mathbf{Z}_M$, *let* $\phi(X) = (x_1, x_2, x_3, \ldots, x_L)$ *where* $x_i \equiv X \pmod{p_i}$ *for all* $i \in \{1, 2, 3, \ldots, L\}$ *then* $X = \phi^{-1}(x_1, x_2, x_3, \ldots, x_L)$ *is described by the following congruence*

$$X \equiv \left\{ \sum_{i=1}^{L} m_i < m_i^{-1} x_i >_{p_i} \right\} \pmod{M}$$

*where* $< \bullet >_p$ *indicates the unary* $\pmod{p}$ *operation.*

The CRT forms the basis for the RNS. In the RNS, two's complement integers are converted to their $L$-tuple residue representation by the ring isomorphism $\phi$ :

$\mathbf{Z}_M \leftrightarrow \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L}$ described by the CRT. The numbers which are in their $L$-tuple representation may be added and multiplied component-wise and reconstructed via the CRT to form the correct result in $\mathbf{Z}_M$. For example, consider the RNS system described by $p_1 = 3$, $p_2 = 5$, and $p_3 = 7$. Then $M = p_1 p_2 p_3 = 105$. Let $a = 7$, and $b = 9$ where $a, b \in \mathbf{Z}_M$. The numbers $a$ and $b$ may be mapped to their RNS 3–tuple representation via the mapping $\phi$:

$$\phi(a) = (< 7 >_3, < 7 >_5, < 7 >_7) = (1, 2, 0)$$
$$\phi(b) = (< 9 >_3, < 9 >_5, < 9 >_7) = (0, 4, 2).$$

Arithmetic may be performed on the RNS $L$–tuple representation of $a, b \in \mathbf{Z}_M$ given by the mapping $\phi$. Let $\phi(a) = (a_1, a_2, a_3, \ldots, a_L)$, and $\phi(b) = (b_1, b_2, b_3, \ldots, b_L)$. Then

$$\phi(a \circ b) = (< a_1 \circ b_1 >_{p_1}, < a_2 \circ b_2 >_{p_2}, < a_3 \circ b_3 >_{p_3}, \ldots, < a_L \circ b_L >_{p_L}),$$

where $\circ \in \{+, -, \times\}$. Consider the 3–tuple representations of $a$ and $b$:

$$(1, 2, 0) + (0, 4, 2) = (< 1 + 0 >_3, < 2 + 4 >_5, < 0 + 2 >_7) = (1, 1, 2) \quad (1.1)$$
$$(1, 2, 0) \times (0, 4, 2) = (< 1 \cdot 0 >_3, < 2 \cdot 4 >_5, < 0 \cdot 2 >_7) = (0, 3, 0). \quad (1.2)$$

For comparison, the mapping of $a + b = 16$ and $ab = 63$ to their RNS 3–tuple representation:

$$\phi(a + b) = (< 16 >_3, < 16 >_5, < 16 >_7) = (1, 1, 2) \quad (1.3)$$
$$\phi(ab) = (< 63 >_3, < 63 >_5, < 63 >_7) = (0, 3, 0) \quad (1.4)$$

The operations performed on the RNS representations of $a$ and $b$ (equations 1.1,1.2) give the same results as the RNS representation of $a + b$ and $ab$ (equations

1.3, 1.4) performed in $\mathbf{Z}_M$. Now consider the restoration of the representation of $a+b, ab \in \mathbf{Z}_M$ from the RNS representations. For $(p_1, p_2, p_3) = (3, 5, 7)$ we have $m_1 = 35$, $m_1^{-1} = 2$, $m_2 = 21$, $m_2^{-1} = 1$, $m_3 = 15$, and $m_3^{-1} = 1$. From above we have $\phi(a+b) = (1, 1, 2)$, and $\phi(ab) = (0, 3, 0)$.

$$\begin{aligned}
\phi^{-1}(1,1,2) &= \left\{\sum_{i=1}^{3} m_i < m_i^{-1}x_i >_{p_i}\right\} \pmod{105} \\
&= \{35 < 2 \cdot 1 >_3 + 21 < 1 \cdot 1 >_5 + 15 < 1 \cdot 2 >_7\} \pmod{105} = 16 \\
\phi^{-1}(0,3,0) &= \left\{\sum_{i=1}^{3} m_i < m_i^{-1}x_i >_{p_i}\right\} \pmod{105} \\
&= \{35 < 2 \cdot 0 >_3 + 21 < 1 \cdot 3 >_5 + 15 < 1 \cdot 0 >_7\} \pmod{105} = 63
\end{aligned}$$

Thus we see that the results produced by the mapping $\phi^{-1}$ are as expected. Generally, the moduli are chosen to be small enough that the adders and multipliers may be implemented in a reasonably small memory-based lookup table. In a VLSI implementation we might leverage advanced memory technology and thereby achieve greater speed and smaller die area.

## 1.2 Complex Residue Number System (CRNS)

The RNS may be used to perform computations with complex numbers by using RNS arithmetic elements to emulate the operations which would be performed using two's complement hardware. The use of RNS arithmetic to perform complex operations is called complex RNS or CRNS. Suppose we have Gaussian integers $a + jb, c + jd \in \mathbf{Z}_M[j]/(j^2+1)$, and $\psi$ denotes the isomorphism between the Gaussian integers and the CRNS: $\psi$: $\mathbf{Z}_M[j]/(j^2+1) \leftrightarrow \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L} \times \mathbf{Z}_{p_1} \times \mathbf{Z}_{p_2} \times \mathbf{Z}_{p_3} \times \cdots \times \mathbf{Z}_{p_L}$. Then

$$(a + jb) + (c + jd) = (a + c) + j(b + d)$$

$$\begin{aligned} &= \psi^{-1}\{\psi(a)+\psi(b)\}+j\psi^{-1}\{\psi(b)+\psi(d)\} \\ (a+jb)\times(c+jd) &= (ac-bd)+j(ad+bc) \\ &= \psi^{-1}\{\psi(a)\psi(c)-\psi(b)\psi(d)\}+j\psi^{-1}\{\psi(a)\psi(d)+\psi(b)\psi(c)\}. \end{aligned}$$

While the complex addition takes only two additions, the complex multiplication takes four multiplications and two additions: the CRNS requires the same number of additions and multiplications as the Gaussian integers.

## 1.3 Quadratic Residue Number System (QRNS)

The QRNS [3, 4] is a variation upon the RNS which allows complex additions to be performed with two RNS additions and complex multiplications to be performed with two RNS multiplications. This enhancement is accomplished by encoding the real and imaginary components into two independent components. Given a prime $p$ of the form $p = 4k+1$ where $k \in$ Z then the congruence $x^2 \equiv -1 \pmod p$ has two solutions in the ring $\mathbf{Z}_p$ that are multiplicative and additive inverses of one another. Let $\hat{j}$ and $\hat{j}^{-1}$ denote the two solutions to the above congruence. Define a mapping $\theta\colon\ \mathbf{Z}_p[j]/(j^2+1) \rightarrow \mathbf{Z}_p \times \mathbf{Z}_p$ by

$$\begin{aligned} \theta(a+jb) &= (z, z^*) \\ z &\equiv (a+\hat{j}b) \pmod p \\ z^* &\equiv (a-\hat{j}b) \pmod p. \end{aligned}$$

Furthermore, the inverse mapping $\theta^{-1}\colon\ \mathbf{Z}_p \times \mathbf{Z}_p \rightarrow \mathbf{Z}_p[j]/(j^2+1)$ is given by

$$\theta^{-1}(z, z^*) = < 2^{-1}(z+z^*) >_p + j < 2^{-1}\hat{j}^{-1}(z-z^*) >_p .$$

Suppose $(z, z^*), (w, w^*) \in\ \mathbf{Z}_p \times \mathbf{Z}_p$. Then the addition and multiplication operations in the ring $< \mathbf{Z}_p \times \mathbf{Z}_p, +, \cdot >$ are given by

$$\begin{aligned} (z, z^*) + (w, w^*) &= (z + w, z^* + w^*) \\ (z, z^*)(w, w^*) &= (zw, z^*w^*). \end{aligned}$$

For example, consider a QRNS system with moduli $p_1 = 5$ and $p_2 = 13$. Let the Gaussian integers $u, v \in \mathbf{Z}[j]/(j^2+1)$ be given as $u = 5+j3$, and $v = 4+j3$. In $\mathbf{Z}_5$ we have $\hat{j}_1 = 2$ and $\hat{j}_1^{-1} = 3$. It can be seen that 2 and 3 are additive and multiplicative inverses of each other in $\mathbf{Z}_5$ and also satisfy the congruence $x^2 \equiv -1 \pmod 5$. In $\mathbf{Z}_{13}$ we have $\hat{j}_2 = 5$ and $\hat{j}_2^{-1} = 8$. Also, $2^{-1} \equiv 3 \pmod 5$, and $2^{-1} \equiv 7 \pmod{13}$. Therefore the QRNS representations of $u$ and $v$ are given by

$$\begin{aligned} \theta(u) &= (z_u, z_u^*) \\ z_u &= (< 5 + \hat{j}_1 3 >_5, < 5 + \hat{j}_2 3 >_{13}) = (1, 7) \\ z_u^* &= (< 5 - \hat{j}_1 3 >_5, < 5 - \hat{j}_2 3 >_{13}) = (4, 3) \\ \theta(v) &= (z_v, z_v^*) \\ z_v &= (< 4 + \hat{j}_1 3 >_5, < 4 + \hat{j}_2 3 >_{13}) = (0, 6) \\ z_v^* &= (< 4 - \hat{j}_1 3 >_5, < 4 - \hat{j}_2 3 >_{13}) = (3, 2). \end{aligned}$$

The arithmetic operations in the QRNS are performed in the same manner as in the RNS. For example:

$$\begin{aligned} \theta(u) + \theta(v) &= (z_u + z_v, z_u^* + z_v^*) = (z_{u+v}, z_{u+v}^*) \\ z_{u+v} &= (< 1 + 0 >_5, < 7 + 6 >_{13}) = (1, 0) \\ z_{u+v}^* &= (< 4 + 3 >_5, < 3 + 2 >_{13}) = (2, 5) \\ \theta(u)\theta(v) &= (z_u z_v, z_u^* z_v^*) = (z_{uv}, z_{uv}^*) \\ z_{uv} &= (< 1 \cdot 0 >_5, < 7 \cdot 6 >_{13}) = (0, 3) \end{aligned}$$

$$z^*_{uv} \;=\; (<4\cdot 3>_5, <3\cdot 2>_{13}) = (2,6).$$

For comparison, note that $uv = 11+j27$ and $u+v = 9+j6$. The QRNS representations of $uv$ and $u+v$ are given as

$$\begin{aligned}
\theta(u+v) &= (z'_{u+v}, z^{*\prime}_{u+v}) \\
z'_{u+v} &= (<9+\hat{j}_1 6>_5, <9+\hat{j}_2 6>_{13}) = (1,0) \\
z^{*\prime}_{u+v} &= (<9-\hat{j}_1 6>_5, <9-\hat{j}_2 6>_{13}) = (2,5) \\
\theta(uv) &= (z'_{uv}, z^{*\prime}_{uv}) \\
z'_{uv} &= (<11+\hat{j}_1 27>_5, <11+\hat{j}_2 27>_{13}) = (0,3) \\
z^{*\prime}_{uv} &= (<11-\hat{j}_1 27>_5, <11-\hat{j}_2 27>_{13}) = (2,6).
\end{aligned}$$

The above results for the QRNS representations $\theta(uv)$ and $\theta(u+v)$ agree with $\theta(u)\theta(v)$ and $\theta(u)+\theta(v)$ computed in the QRNS representation. The isomorphism $\theta$ is generally implemented by a combination of arithmetic elements and table lookup. Since the $z$ and $z^*$ channels are independent we are able to easily construct parallel hardware to perform operations on both channels at the same time without any communication between the channels. This parallelism allows us to easily perform a complex addition or multiplication in one cycle. While parallel hardware would allow us to perform a CRNS addition in one cycle, the multiplication in the CRNS requires two additions and four multiplications. Using the same amount of hardware as a QRNS multiplier-accumulator, a CRNS multiplier-accumulator would take twice as many cycles to complete a single multiply-accumulate operation.

## 1.4 Galois Enhanced QRNS (GEQRNS)

The QRNS requires us to implement a multiplier which takes $N$ bit inputs and produces an $N$ bit output. The multiplier could be implemented using either a direct implementation with modular correction or a lookup table. The primary disadvantage of this is that despite the small size of the RNS adder, the multiplier is still large. We may take advantage of the properties of Galois fields [5] to simplify the implementation of an RNS multiplier.

For any prime modulus $p$ there exists some $\alpha \in \mathbf{Z}_p$ that generates all non-zero elements of the field $GF(p)$. That is to say $\{\alpha^i \mid i = 0, 1, 2, \ldots, p-2\} = GF(p) \setminus 0$. Thus, we may uniquely represent all non-zero elements of $\mathbf{Z}_p$ by their exponents. These number theoretic logarithms may be added modulo $p-1$ to produce multiplication: $\alpha^{<i+j>_{p-1}} =< \alpha^i \alpha^j >_p$. Note that since zero is not an element of $GF(p) \setminus 0$ the zero must be handled as an exception. Practically, this means that the inputs must be checked before the number theoretic logarithm to determine whether either one is a zero, and if one of the inputs is a zero, then the output of the multiplier should be set to zero.

For example, suppose that $p = 7$. Then $\alpha = 3$ generates $GF(7) \setminus 0$: $\{3^i \mid i = 0, 1, 2, 3, 4, 5\} = \{1, 3, 2, 6, 4, 5\}$. Suppose we wish to multiply 2 and 3. First we would take the number theoretic logarithm of 2 and 3 to the base $\alpha = 3$:

$$\log_3(2) = 2 \iff 3^2 \equiv 2 \pmod 7$$
$$\log_3(3) = 1 \iff 3^1 \equiv 3 \pmod 7.$$

In order to multiply 2 and 3 we now add the number theoretic logarithms modulo $p-1$:

$$2 \cdot 3 =< 3^2 \cdot 3^1 >_7 =< 3^{<2+1>_6} >_7 =< 3^3 >_7 = 6.$$

The architecture of a GEQRNS multiplier is illustrated in Figure 1.1 without the zero detection and handling indicated. The multiplier requires two duplicate $N$-entry memories to perform the number theoretic logarithm, and an $N+1$-entry table to perform the modulo $p-1$ correction and number theoretic exponentiation. Note that while the modulo $p-1$ correction and number theoretic exponentiation represent two separate steps, they may be integrated into a single table. Typically, the multiplicands will be converted to the GEQRNS number theoretic logarithm form by the conversion engine which computes the residues of the integer inputs.

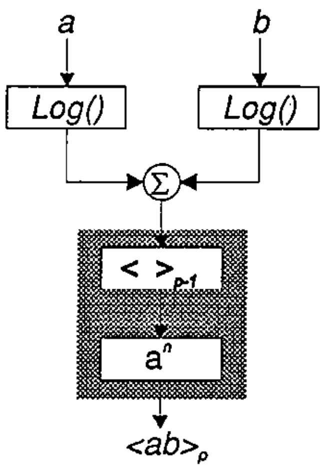


Figure 1.1: Block Diagram of a GEQRNS Multiplier

## 1.5 L-CRT

The $L$-CRT [1, 2] offers an alternative to the CRT which has the advantage of integrating scaling into the CRT and avoiding the need for a modulo $M$ adder. The $L$-CRT is computed by factoring $M$ into a real scale factor $V$ and an integer $M' = 2^k$, where $k \in \mathbf{Z}^+$, such that $M = VM'$, and $0 < M' < M$. Additionally, as for the

CRT, $m_i = M/p_i$. The $L$-CRT is given as

$$X_S = \left\{ \sum_{i=1}^{L} \lfloor m_i < m_i^{-1} x_i >_{p_i} /V \rfloor \right\} \pmod{M'},$$

where $\lfloor \bullet \rfloor$ denotes the least integer or floor function. Since $M' = 2^k$ where $k \in \mathbf{Z}^+$ we may compute the sum $X_S$ using regular $k$–bit two's complement adders. The $\lfloor m_i < m_i^{-1} x_i >_{p_i} /V \rfloor$ term for any fixed set of moduli is dependent only upon $x_i$ and thus may be generated using a small, fast memory based table lookup. The disadvantage of the $L$-CRT is that it may introduce an error into the computed $X_S$. The error in the $L$-CRT is given by $0 \leq |X/V - X_S| < L$. For front-end signal processing applications this error is not critical since $L \ll M$. A block diagram of two $L$-CRT engines is shown in Figure 1.2.

The $L$-CRT has the advantage of avoiding the modulo $M$ adder required to implement the 'true' CRT and provides a means of scaling without additional hardware. For VLSI and discrete implementations this advantage is particularly important since division, like multiplication, are space-time intensive and cannot be performed in the RNS since it is division-free.

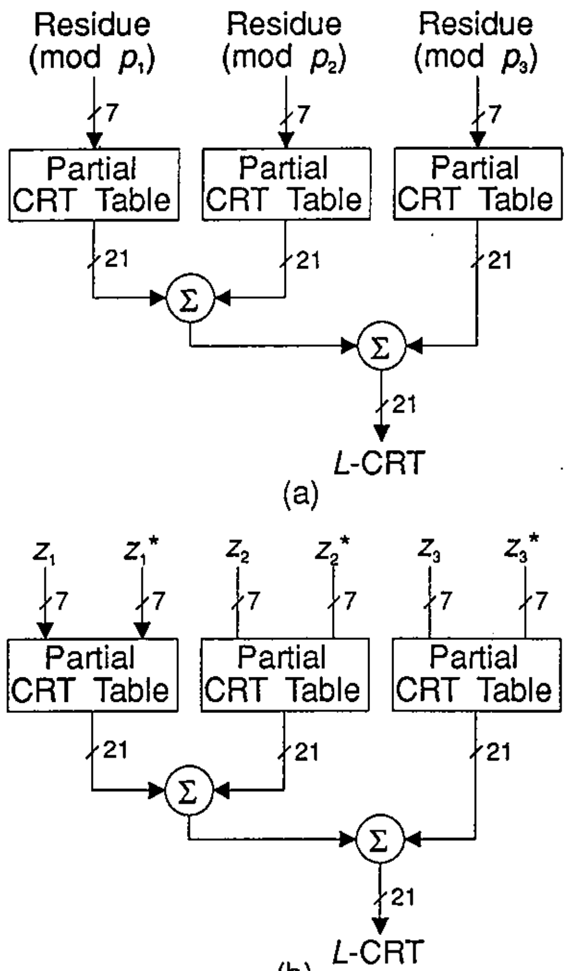


Figure 1.2: Block Diagram of $L$-CRT (a) and QRNS Augmented $L$ CRT (b)

# Part II

# InvestiGATOR Array Processor Backplane

Chapter 2

# INTRODUCTION

## 2.1 Motivation

There exists a need for an environment appropriate to the task of developing experimental array processors. This need is indicated by the large I/O requirements and physical size of experimental array processors. Traditional environments such as personal computers or larger systems such as the VME bus are not appropriate as they lack adequate space and I/O capabilities. Thus the motivation is established for the development of a testbed for experimental array processors.

Additional capabilities are desirable. In particular, beyond the need to solve physical form factor problems and I/O bandwidth bottlenecks, there is an additional desire that the system should be host independent. The ideal host interface for achieving host independence is the SCSI interface. The SCSI interface exists on all common personal computers and workstations. There also exist a number of peripherals which may take advantage of the SCSI interface, thus allowing the testbed to utilize a number of mass storage and data acquisition products.

## 2.2 Design Parameters

Given the motivation presented in the previous section, the design parameters are described as follows. The control of the array processor and the SCSI interface require substantial machine intelligence. Thus the selection of a microprocessor is required.

The Motorola 68030 was selected since it is capable of sustaining block data moves of approximately forty megabytes per second (at 20 MHz), and because of previous design experience with the 68000 family. First generation SCSI controller chips such as the NCR 8350 require substantial processor intervention in order to operate: each byte transferred causes an interrupt to occur. Additionally, these first generation SCSI controller chips were only capable of asynchronous operation at data rates of approximately 1.6 megabytes per second while many hosts operate synchronously at a maximum data rate of five megabytes per second. A second generation device was selected, the Western Digital 33C93A. The WD33C93A (second sourced by Advanced Micro Devices and sometimes referred to as the Am33C93A) executes SCSI commands independently of the host processor and is capable of transmitting large quantities of data without host intervention. For purposes of debugging, the array processor testbed also features RS-232C serial communications.

Memory requirements for the testbed are modest. The testbed need only buffer data transactions between the host and array and perform some translation of commands from the host to the array. Thus it was determined that the testbed processor would only require one megabyte of high speed RAM and 128 kilobytes of ROM. Since the processor typically is moving large, contiguous blocks of data between the SCSI processor and the array processor, a memory architecture which performs well in block operations is desirable. A dynamic RAM variant called static-column RAM (SCRAM) is particularly well suited to this task. The SCRAM is fundamentally a standard DRAM, however, once the row address has been latched into the device, the device operates as a static RAM for all subsequent accesses to that row of memory. These accesses may occur until refresh is required. The advantage to this means of memory operation are that a 70 ns device offers 35 ns access times during static column operation. The static column mode of operation is synergistic with the 68030's burst mode of operation. Using the burst mode of operation the 68030 may read four longwords with reduced penalty. In particular, in the burst mode of operation, the worst-case first word read time is two clock cycles (at 20 MHz, $T_{cycle} = 50$ns). Subsequent accesses in non-burst mode still execute in two clock cycles. Subsequent burst-mode accesses execute in one clock cycle. Thus, the maximum memory bandwidth without burst access is forty megabytes per second, while the maximum memory bandwidth with burst access is sixty-four megabytes per second.

# Chapter 3

# IMPLEMENTATION

This chapter describes the implementation of the InvestiGATOR array processor testbed. The description is broken into modules reflecting the various major components of the backplane: the CPU, the memories, the I/O components, the array interface, and remaining miscellaneous material.

## 3.1 Architecture

The InvestiGATOR backplane and SCSI control processor is constructed from several discrete blocks. These blocks may be divided into four groups. The first, the CPU is based upon the Motorola MC68030. The second, the memory, consists of one megabyte of high performance static-column RAM, and 128 kilobytes of low performance EPROM. The third group is the I/O module which includes a high performance SCSI port, dual RS-232C serial ports, and an I/O expansion port. The fourth group is the array bus and interface. A block diagram of the InvestiGATOR is shown in Figure 3.1.

The SCSI port is a single-ended, eight-bit implementation supporting synchronous transfers up to five megabytes per second. The SCSI port has a local thirty-two kilobyte buffer which allows the central processor to operate without interference while transfers are underway. SCSI packets may be transferred either to or from the InvestiGATOR with as few as two interrupts of the central processor. This autonomous operation allows the CPU to dedicate a large percentage of its processing

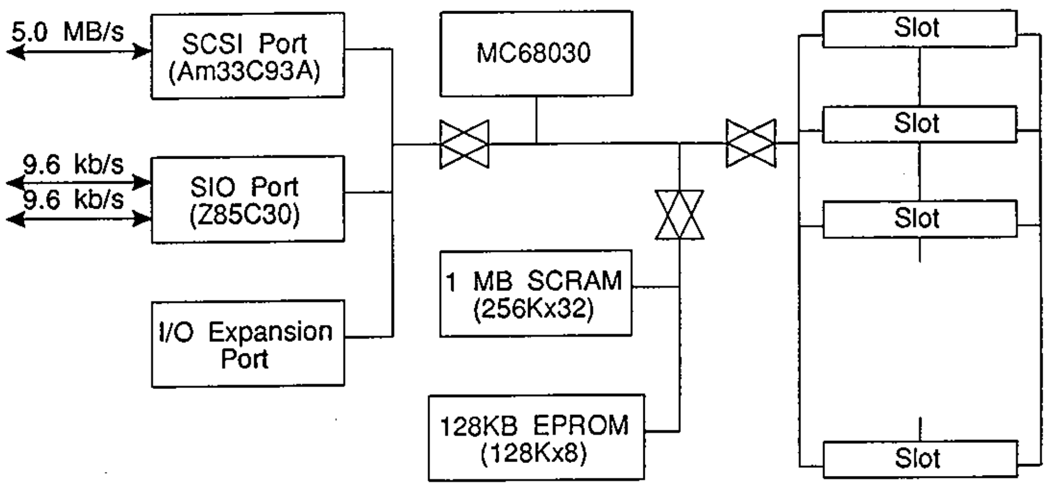


Figure 3.1: Block Diagram of the InvestiGATOR Array Processor Testbed

budget to servicing the attached experimental array processor.

The serial port supports two RS-232C channels with programmable baud rates of up to 9600 bps. The serial port is intended to act primarily as a debugging tool. The I/O expansion port has a full thirty-two bit data bus, twenty-bit address bus, and interrupt capabilities. This bus may be used to attach data acquisition, additional I/O capabilities, or memory.

The RAM block is based upon static-column RAM supporting synchronous and burst-mode accesses. This memory offers very high performance in block transfers.

## 3.2 CPU Module

This section describes the generation of the various signals which are used in the CPU module to service the MC68030, and signals which are used to interface with external devices and busses. This section refers to schematics which are found in Appendix A. A block diagram of the CPU module with its major subsystems is shown in Figure 3.2.

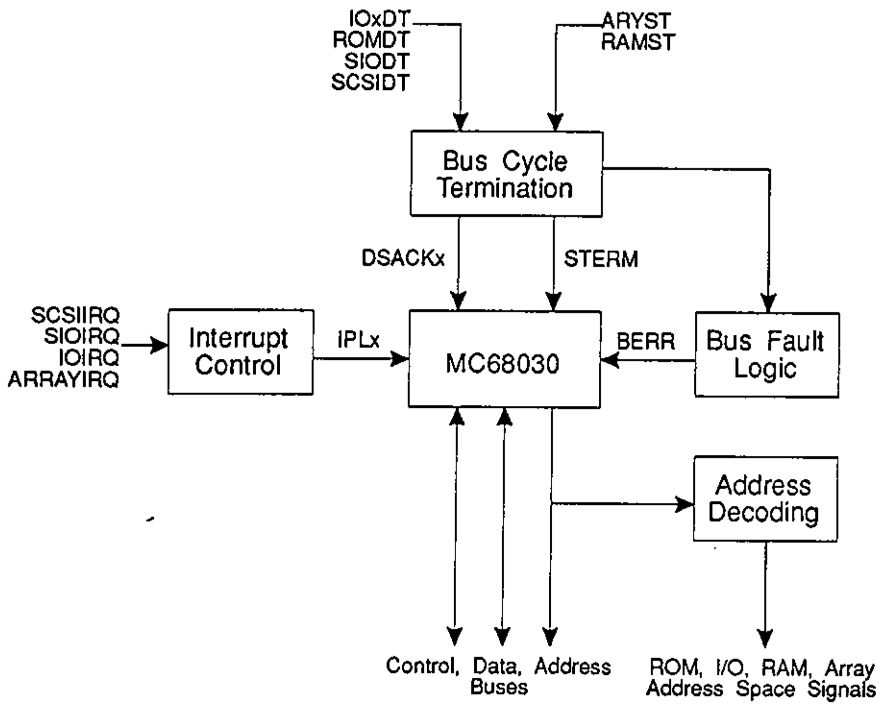


Figure 3.2: Block Diagram of CPU Module

### 3.2.1 Cache Control

The MC68030 provides a mechanism whereby external circuitry may indicate to the 68030 which addresses are cachable, the cache inhibit input, CIIN*. CIIN* is generated by PAL0 and inhibits the cache when accessing the I/O and array addressing spaces. Additionally, the 68030 provides a means for disabling the cache from external hardware, primarily for debugging purposes. This is the cache disable input, CDIS*. CDIS* may be asserted or negated using switch S3.

The primary reason for the selection of the MC68030 as the control processor of the InvestiGATOR was its on-chip instruction/data cache and burst cache fill mechanism. The 68030 provides a means of bursting four longwords of instructions or data into the cache. This is accomplished using the MC68030's cache burst request/acknowledge (CBREQ*/CBACK*) handshaking protocol. When the 68030 runs a bus cycle in which it can execute a burst fill of the cache it asserts the CBREQ* signal. If the addressed device wishes to proceed with a burst fill of the cache it must acknowledge the burst request with CBACK*. In a zero wait state system the 68030 can read four longwords in eight cycles (*i.e.*, forty megabytes per second) using standard bus cycles while the same four longwords can be read in five clock cycles (*i.e.*, sixty-four megabytes per second) using burst mode. There is support for burst filling of the cache from the RAM module only (see Table 3.2). A burst acknowledge on the part of the RAM module is passed through a D flip-flop clocked 180 degrees out of phase with the 20 MHz system clock in order to stretch the CBACK* signal.

### 3.2.2 Interrupt Control

The MC68030 provides a seven level prioritized interrupt mechanism using the IPL0-2* signals. PAL1 provides priority encoding of the various interrupt signals generated in the InvestiGATOR. The majority of the signals are provided to I/O devices, however, there is also an interrupt line reserved for the array bus. The prioritization of the interrupt sources is given below:

| Request Priority | Description |
|---|---|
| 7 | NMI (Non-Maskable Interrupt). Reserved. |
| 6 | SCSI Port. |
| 5 | Reserved. |
| 4 | SIO port. |
| 3 | Reserved. |
| 2 | I/O Bus. |
| 1 | Array Bus. |

Table 3.1: Interrupt Priority Levels

The InvestiGATOR uses the MC68030's interrupt autovector mechanism to vector interrupts. This is accomplished by asserting the AVEC* input of the 68030 when an interrupt acknowledge cycle is executed. AVEC* is generated by PAL0 using a clocked output. AVEC* is asserted when PAL0 detects an interrupt acknowledge cycle. The 68030 also provides one additional signal related to interrupts, the IPEND* (interrupt pending signal). The IPEND* signal is not used by the InvestiGATOR.

### 3.2.3 Address Space Decoding

Address space decoding is provided by PAL0. PAL0 decodes four primary address spaces: RAM space, ROM space, I/O space, and array space. These address space signals are address strobe qualified. This address space arrangement consumes sixty-four megabytes of the four gigabyte available address space, however, the sixty-four megabyte space is repeated (*i.e.*, A26-A31 are ignored). Accesses to memory spaces besides program and data space are ignored (with the exception of interrupt acknowledge cycles which run in CPU space) and will result in a bus fault after a timeout. Address space decoding is summarized in Table 3.2.

| | Address Range | Description |
|---|---|---|
| C | 0h—1FFFFh | ROM space. |
| | 20000h—FFFFFh | I/O space. |
| B,C | 100000h—FFFFFFh | RAM space. |
| | 1000000h—3FFFFFFh | Array space. |

C=cachable, B=burst cycle support.

Table 3.2: Address Space Decoding

### 3.2.4 Bus Cycle Termination

The 68030 provides two mechanisms for normal termination of bus cycles: asynchronous termination and synchronous termination. Both means of termination are supported by the InvestiGATOR. The synchronous termination mechanism is a high speed termination mechanism for use with thirty-two bit data ports only. In practice, only the RAM space and array space use synchronous termination. The MC68030's synchronous termination input, STERM* is generated by taking the logical OR of the two possible sources of synchronous termination requests, and then using a D flip-flop clocked 180 degrees out of phase with the 20 MHz system clock to stretch the STERM* signal, see Figure 3.3

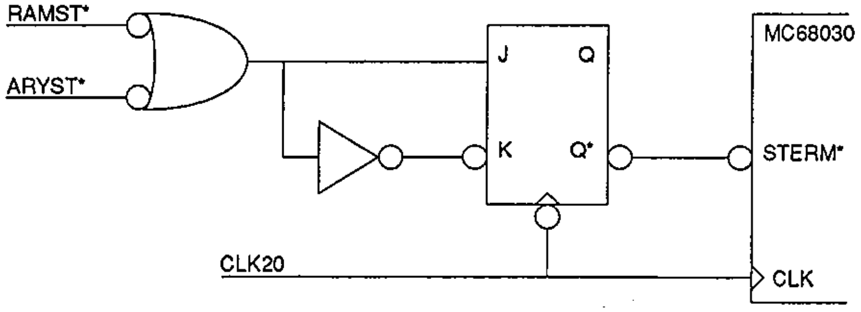


Figure 3.3: STERM Signal Input and Conditioning

The asynchronous bus cycle termination mechanism allows for dynamic bus sizing for eight, sixteen, and thirty-two bit ports. The asynchronous termination signals, DSACK0* and DSACK1*, are provided by PAL1A which generates the appropriate DSACKs for various ports (primarily I/O).

### 3.2.5 Abnormal Bus Cycle Termination: Bus Error Control

It is possible to attempt to access addresses for which there is no corresponding device. In this event it is necessary for external circuitry to terminate the bus cycle.

Additionally, it may be desirable to terminate an I/O or array bus cycle with an error condition. Bus cycles may be terminated with a fault condition by assertion of the MC68030's BERR* signal. Assertion of the BERR* signal is controlled by the BERR control state machine, located on MACH2. This state machine tracks bus cycles and asserts BERR* when the I/O or array busses request, or in the event of a timeout, indicated by the trickle count output of an eight bit watchdog timer (counter). A state machine diagram is given in Figure B.1.

### 3.2.6 Byte Select Signals

The CPU module provides byte select signals (UU*, UM*, LM*, and LL*) to external modules by decoding the A0, A1, SIZ0, and SIZ1 outputs of the 68030. These byte selects are decoded by PAL1A and are not qualified by the address strobe.

### 3.2.7 Miscellaneous Signals

The InvestiGATOR does not support multiple bus mastering in the controller so the BR* (bus request) input is negated. The BG* (bus grant) signal is ignored and the BGACK* (bus grant acknowledge) signal is negated. The MC68030's memory management unit may be disabled using the MMUDIS* input to the 68030. Access to this signal is provided using switch S4.

## 3.3 Memory Module

This section describes the operation of the RAM and ROM modules. The RAM architecture is based upon a single thirty-two bit wide bank of 70 ns static column RAM (SCRAM) with a capacity of one megabyte. The SCRAM controller is based upon a high density PLD, the AMD Mach 110, with high resolution timing generated by the AMD Am2971A programmable event generator (PEG). The ROM architecture is based upon a single eight-bit wide bank of EPROM with a capacity of 128 kilobytes. The ROM is only intended for SCSI control processor diagnostic and operating code. Time critical code sections are moved from the ROM to the main memory, the SCRAM. Microcode and data may be loaded from the host after boot. In situations where the InvestiGATOR is being used as a standalone data collection unit microcode might be loaded from a non-volatile semiconductor disk resident on the I/O bus.

### 3.3.1 Static Column RAM

The InvestiGATOR contains a one megabyte bank of SCRAM. The SCRAM is used as an alternative to standard DRAM because of its high speed access properties: sequential accesses to the same column proceed substantially faster than an access to the same speed rated standard DRAM. The SCRAM achieves no-wait-state operation when operating in static column mode. This is an attractive property when coupled with the 68030's burst mode and when one considers that the primary use for this bank of RAM will be to perform SCSI block transfers.

There are penalties to pay for the high performance of the SCRAM: SCRAM is fifty per cent to one-hundred per cent more expensive than standard DRAM, SCRAM requires significantly more control logic than standard DRAM, and in the event of a non-static column mode access, there is a substantial penalty to pay in cycling a new row address. However, given the design constraints, the static column architecture is the best solution.

The SCRAM architecture is composed of several components. There is the SCRAM itself, data transceivers, address multiplexer, address comparator, burst counter, refresh timer, high-time resolution sequencer, and byte select decoder. A block diagram of the SCRAM architecture is shown in Figure 3.4.

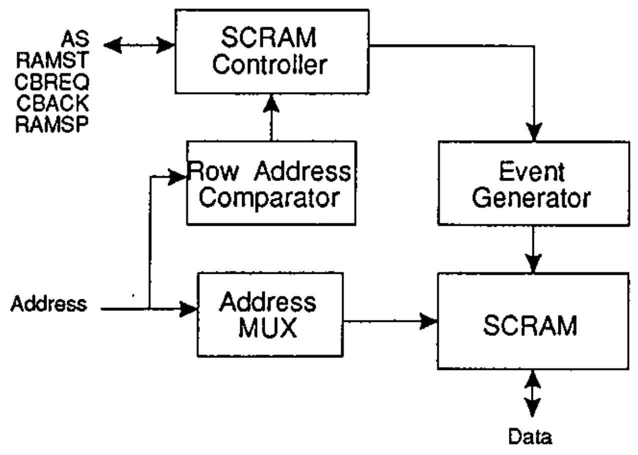


Figure 3.4: Block Diagram of SCRAM Architecture

The burst counter serves to cycle the two lowest order bits of the address during burst accesses. For example, if an access is a miss in the 68030's internal cache, caching is allowed, and the target of the access supports burst mode accesses then in order to keep latency (from the execution unit's point of view) minimal the required word is read. Then the next longword address, modulo four, is read, and so on until four longwords have been read. The burst counter is integrated onto the PLD which contains the controller state machine.

The address comparator serves to allow the controller to determine whether an access is a static column hit. The address comparator contains both a register and a comparator so that the previous row address can be stored for comparison with future accesses. Note that refresh cycles do not invalidate the register contents of the address comparator. Validity of the contents of the address comparator is controlled by the state of the RAS signal: the contents (and thus the output) of the address comparator are valid if and only if RAS is asserted.

The refresh counter is a simple eight bit counter whose trickle-count output sets a refresh request to the controller state machine. The refresh counter issues a refresh request 256 cycles ($T$=50 ns) after it is reset for a net of one request every 12.8 ms resulting in each of the 512 rows of RAM being refreshed every 6.6 ms, meeting the required 8 ms refresh cycle period.

The controller issues commands to the sequencer to perform operations on the RAM. The sequencer is an AMD Am2971A programmable event generator (PEG) which is capable of generating sequences of signals with 10 ns timing resolution. Some of the signals are routed directly to their targets while others are routed through a PLD which provides byte select coding, primarily for write operations. Additionally the controller handles all handshaking with the CPU. The state machine must handle a number of conditions:

- Refresh cycle
- Static column miss, read without burst
- Static column miss, read with burst
- Static column hit, read without burst
- Static column hit, read with burst
- Static column miss, write
- Static column hit, write

Examining the controller state machine diagram (see Figure B.3) we see that the state machine implements the read sequences using a variety of shared state sequences. By sharing state sequences we arrive at a much more efficient implementation of the controller state machine.

The following refers to Figure 3.5. The static column RAM device is dependent upon four control signals: chip select (CS), row address strobe (RAS), write strobe (WR), and output enable (OE). RAS, WR, and OE are generated by the PEG and fed directly to the SCRAM devices while the CS signal is generated by the PEG it is subject to byte select coding by PAL4 using the byte select signals (UU, UM, LM, LL) generated by the CPU module. The data lines are buffered using four Am29C861A CMOS bus transceivers under the control of the SCRAM controller state machine. The address lines are multiplexed by a pair of Am29C827A bus drivers acting as a row/column address multiplexer under control of the controller via the PEG.

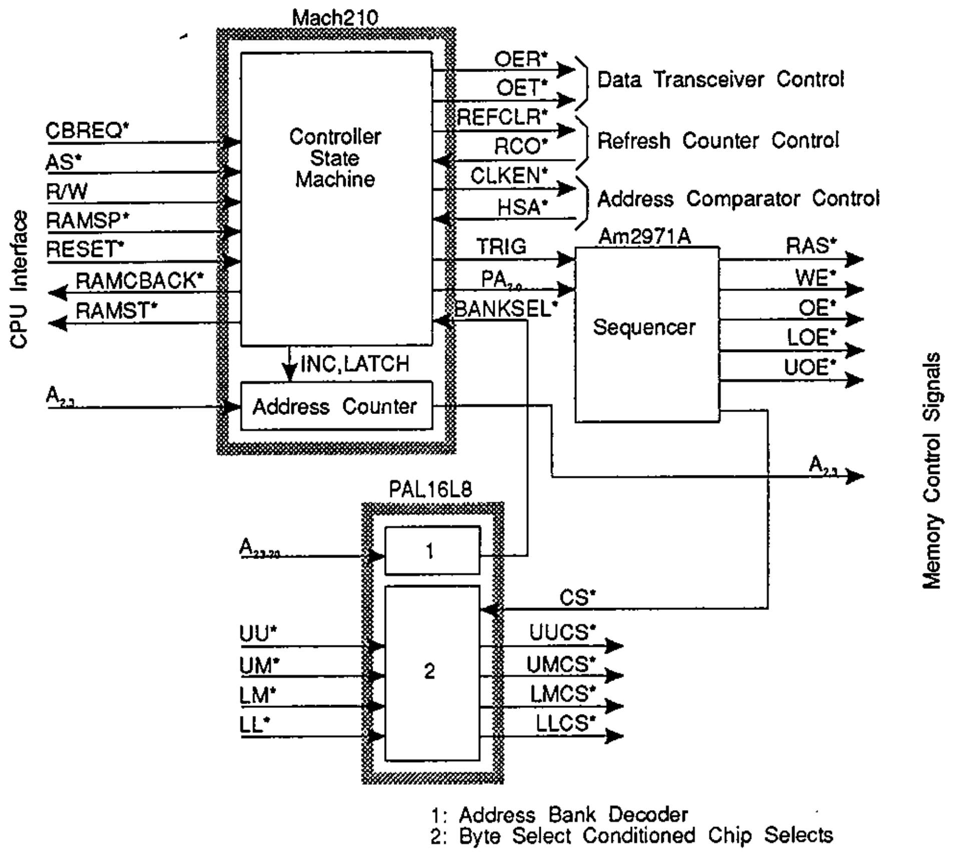


Figure 3.5: SCRAM Controller Architecture

The refresh counter operates in a free counting mode, driven by the 20 MHz system clock. Two-hundred fifty-six clock cycles (12.8 ms) after a counter reset the trickle-count output (RCO) is asserted for one clock cycle which in turn sets the refresh request SR flip-flop in the controller state machine PLD. After the completion of the current memory transaction the controller resets the counter and the refresh request SR flip-flop via the CLRREF signal and orders the PEG to execute a hidden refresh cycle. Under worse case conditions a refresh request could suffer a response latency of up to twelve clock cycles (600 ns). Thus, under these worse case conditions a hidden refresh cycle might be executed every 13.4 ms implying a refresh of every row of the SCRAM every 6.9 ms, still within the required 8.0 ms.

The controller handshakes with the CPU module via the AS, CBREQ, RAMSP, Read/Write, CBACK, and STERM signals. The R/W, AS and RAMSP signals are used in conjunction with additional address decoding provided by PAL4 (via the BANKSEL signal from PAL4) to initiate memory transactions. The CBREQ/CBACK handshaking pair is used to control burst cycles.

The controller orders the PEG to execute sequences using the PA2-0 and TRIGx outputs. The PA2-0 signals provide an address to the PEG to determine the starting point in its memory for execution while the TRIGJ/TRIGK outputs are fed through a negative- edge triggered flip-flop to generate a trigger signal which will arrive at a time when the PEG address inputs (PA2-0) are guaranteed valid and cause the PEG to begin execution with minimal latency. The chip select signals generated by the PEG are gated using the byte selects generated by the CPU module with controller override via the CSALL signal. The PEG also generates the RAS, WR, and OE signals used by the SCRAM. Additionally, the PEG controls the address multiplexer via the AREG signals which control the output enables of the address drivers.

The address comparator is a combination register/comparator. The controller causes the comparator to latch a new row address using the CLKEN signal. When the address comparator determines that the row address at its input matches that stored in its internal register it signals the controller using the HSA signal. Finally, the data transceivers are controlled by the OER and OET controller signals.

The burst address counter is integrated into the controller PLD. This counter is a simple two-bit counter with load and increment controls from the controller state machine, load inputs AI1,0, and AO1,0. Negation of the load or latch and increment controls implies a hold state. The outputs of this counter are fed through the address multiplexer to the SCRAM array. Note that since the least significant bits of the column address are fed through the burst address counter, the presentation of a new column address to the SCRAM array is limited by both the address multiplexer and the speed with which the address counter can latch a new address and present it to the address multiplexer.

The SCRAM must be verified each time the power is applied. There are standard algorithmic test methods which facilitate functional testing of the DRAM and detection of common faults [6]. The standard test methods discussed in [6] are targeted primarily at functional testing of DRAMs in VLSI testers, not testing of the memory in circuit. These methods may be adapted with the addition of tests to exercise the surrounding architecture. In particular, during testing of the first InvestiGATOR board, a stuck-at fault (SAF) was discovered in one of the address multiplexer buffers. A test to find SAFs in the address multiplexer buffers is given in Figure 3.6. Once the address multiplexers are verified the data transceivers should be verified. Note that malfunctioning data transceivers could potentially mask or simulate an address multiplexer SAF, thus, special precaution should be taken in the implementation of the address multiplexer SAF detection so as not to cause an erroneous conclusion as to the status of the address multiplexers.

```
for i=0 to n-1
  M[0]:=0
  M[2^i]:=1
  if M[0]!=0 then there exists an SA0 fault @ bit i
  M[0]:=1
  M[2^i]:=0
  if M[2^i]!=1 then there exists an SA1 fault @ bit i
end
```

Figure 3.6: Pseudo-Code for Address Multiplexer SA Fault Detection

Once the status of the surrounding architecture is verified, [6] suggests that tests for unlinked SAFs, unlinked transition faults (TFs), unlinked coupling faults (CFs), linked CFs, linked CFs and TFs, address decoder faults (AFs), and various pattern sensitive faults (PSFs) be conducted. It turns out that two tests will provide fault coverage for SAFs, TFs, AFs, linked CFs, linked TFs, unlinked idempotent, and unlinked inversion CFs: the March C and March B algorithms.

Each march element of a march sequence consists of an arrow pointing up or down, indicating the direction of march in address space, and a sequence of read and write operations. For example, $\Uparrow$ indicates an address sequence from zero to $n-1$, while $\Downarrow$ indicates an address sequence from $n-1$ to zero. The March C algorithm is given in Figure 3.7. The March B algorithm is given in Figure 3.8. Both the March C and March B algorithms assume that an initial $\Uparrow$(w0) march is executed to initialize the memory before the test algorithm is executed.

The most common PSFs which occur are neighborhood pattern sensitive faults (NPSFs). NPSFs are faults where the writing of memory cells adjacent to a base cell will cause an unwanted transition in the base cell. The cells most likely to effect a

$\{ \Uparrow(r,w1); \Uparrow(r,w0); \Uparrow(r); \Downarrow(r,w1); \Downarrow(r,w0); \Downarrow(r); \}$

Figure 3.7: March C Algorithm for Memory Testing

$\{ \Uparrow(r,w1,r,w0,r,w1); \Uparrow(r,w0,w1); \Downarrow(r,w0,w1,w0); \Downarrow(r,w1,w0); \}$

Figure 3.8: March B Algorithm for Memory Testing

base cell – and thus expose an NPSF – are the four cells adjacent to the base cell in the north, south, east, and west directions. A basic NPSF detection algorithm, suggested by [6] is given in Figure 3.9.

```
write all base cells with zero;
for each base cell
  apply a pattern;
  read base cell and compare against expected value (zero);
end;
write all base cells with one;
for each base cell
  apply a pattern;
  read base cell and compare against expected value (one);
end;
```

Figure 3.9: A Basic NPSF Detection Algorithm

### 3.3.2 ROM Controller and Architecture

The InvestiGATOR contains a single bank of 128K × 8-bit wide (128 kilobytes) EPROM. This ROM is a low performance memory which contains basic firmware for the InvestiGATOR and may contain some firmware for the array under test. ROM read cycles are executed in three clock cycles yielding a net bandwidth of 6.67 megabytes per second. Code segments demanding higher performance may be shadowed to the RAM space.

## 3.4 I/O Bus and Devices

The InvestiGATOR supports an I/O bus through which it communicates with the outside world. Currently the I/O bus contains a SCSI controller, and a serial (RS-232C) port. The SCSI controller utilizes the Western Digital 33C93A SCSI bus controller chip and contains a thirty-two kilobyte data buffer. The serial I/O controller uses the AMD Z85C30 ESCC (Enhanced Serial Communications Controller) to provide two channels of RS-232 I/O. Allowances are made for the addition of peripherals to the InvestiGATOR's I/O bus. Some of the allowances include a wired-OR interrupt request line and three data transfer acknowledge lines: one for each size data port supported by the MC68030. The accessibility of the I/O bus is intended to compensate for the potential unavailability or unsuitability of a SCSI bus equivalent peripheral.

### 3.4.1 SCSI

The SCSI port is built around the Western Digital 33C93A SBIC (SCSI Bus Interface Chip). The SCSI port is designed to use a form of I/O called DBA (direct buffer access ) for data block transfers. Using DBA, the SBIC performs block transfers directly to and from a thirty-two kilobyte local buffer memory without processor intervention. This allows the SBIC to achieve its rated five megabyte/second data transfer rates and allows the control processor to avoid the performance penalties associated with interrupt servicing overhead. A block diagram of the SCSI port architecture is depicted in Figure 3.10.

The SBIC operates in two modes during normal operation in the InvestiGATOR: direct addressing mode and DBA mode. In the direct addressing mode the

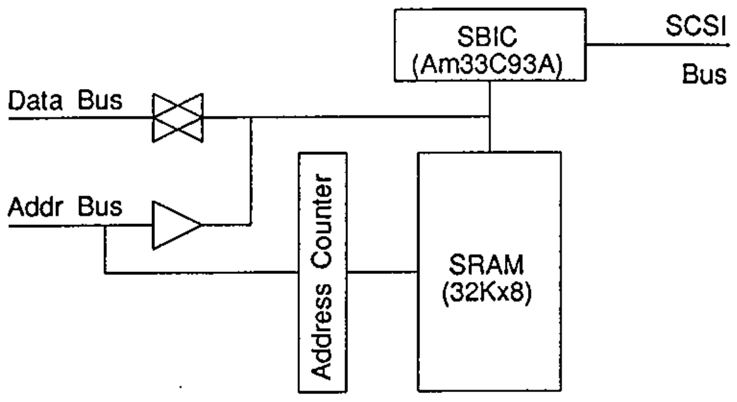


Figure 3.10: Block Diagram of the SCSI Port

processor performs transactions with the SBIC by using hardware assisted time multiplexing of the address and data to the SBIC address/data port. Direct addressing mode contrasts with indirect addressing mode where the processor first would write an address to the SBIC and then the next SBIC access would be performed on the register whose address was written in the previous cycle. Indirect addressing mode carries obvious penalties since two real accesses are required for every data transaction. The SBIC normally is kept in a DBA stand-by mode: that is, whenever the processor is not accessing the SBIC or RAM buffer the SBIC is in DBA mode. When the processor attempts to perform a transaction with the SBIC or RAM the SBIC is switched out of DBA mode so that the transaction may proceed.

In DBA mode the SBIC has control of the RAM buffer. Reads and writes are accomplished using the SBIC read enable and write enable signals. Since the SBIC has no means of handshaking with external logic when performing individual transactions with the buffer RAM, it is up to the control architecture to ensure that the transaction meets the SBIC's timing requirements. Additionally, the SBIC provides no direct control of the address counter; rather, the control of the counter is implicit. After each buffer read or write operation, the counter must be incremented by the external hardware. The control logic determines when to increment the counter by observing the read and write strobes. Address counter control in DBA mode is performed by observing the RE and WE strobes which are controlled by the SBIC in this mode.

The SCSI-2 specification gives a list of commands which a processor on the SCSI bus can implement. Some of the commands listed are optional while others are mandatory under the SCSI-2 specification. A table of these commands and whether the InvestiGATOR responds to the commands is given in Table 3.3.

| | Command Name | Notes |
|---|---|---|
| O | Change Description | Not Implemented. |
| O | Compare | Not Implemented. |
| O | Copy | Not Implemented. |
| O | Copy and Verify | Not Implemented. |
| M | Inquiry | |
| O | Log Select | |
| O | Log Sense | |
| O | Read Buffer | Used to read program memory and control store. |
| O | Receive | Used to transmit command and data packets to InvestiGATOR. |
| O | Receive Diagnostic Results | Used to retrieve diagnostic results. |
| M | Request Sense | |
| M | Send | Used to receive command and data packets from InvestiGATOR. |
| M | Send Diagnostic | Used to request diagnostics to be performed. |
| M | Test Unit Ready | |
| O | Write Buffer | Used to load program memory and control store. |

O=optional, M=mandatory, according to SCSI-2 definition.

Table 3.3: SCSI-2 Command Set

### 3.4.2 SIO

The serial I/O interface is provided for software development and diagnostic purposes. The serial controller is based upon an AMD Z85C30 Enhanced Serial Communications Controller (ESCC). The ESCC-I/O bus interface is composed simply of an eight-bit buffer and a PAL-based controller.

The ESCC supports two channels of serial communications and independent baud rate generation. Two channels of serial I/O are supported by the InvestiGATOR since the additional cost is minimal. In the case of the InvestiGATOR the baud rate is generated by dividing down the 10 MHz system clock to the appropriate baud rate. The baud rate is programmed by providing a time constant for each channel. The time constants appropriate to some common baud rates assuming $f_{CLK}$=10 MHz, and a clock multiplier of sixteen are provided in Table 3.4.

| Desired Baud | Time Constant | Actual Baud | Per Cent Difference |
|---|---|---|---|
| 300 | 1044 | 299.904 | -0.032 |
| 1200 | 262 | 1201.92 | 0.159 |
| 2400 | 132 | 2403.85 | 0.158 |
| 4800 | 67 | 4807.69 | 0.155 |
| 9600 | 35 | 9469.70 | -1.296 |
| 19200 | 18 | 19531.3 | 1.510 |

Table 3.4: Time Constants versus Baud Rates for Enhanced Serial Communication Controller

The ESCC's registers are mapped in I/O space as described in Table 3.5.

The serial ports are brought out to DB9 connectors on the back of the InvestiGATOR. The signals are translated via the RS-232C level compatible MC1448 transmitter and MC1449 receiver. This transmitter/receiver pair was chosen for its robustness. The pinout of the InvestiGATOR's serial ports is non-standard and de-

| Address | Description |
|---|---|
| 20800h | Channel B Control Register. |
| 20801h | Channel B Data Register. |
| 20802h | Channel A Control Register. |
| 20803h | Channel A Data Register. |

Table 3.5: Enhanced Serial Communication Controller Register Memory Map

picted below in Figure 3.11.

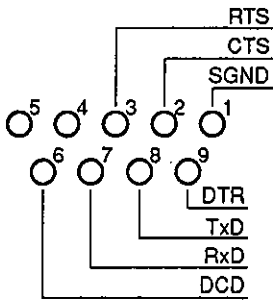


Figure 3.11: InvestiGATOR Serial Port Pinout

A cable suitable for connecting the InvestiGATOR to an IBM PS/2 host was constructed according to the diagram in Figure 3.12. The cable is suitable for XON/XOFF flow-control protocol and is not suitable for hardwire (*i.e.*, REQ/ACK or RTS/CTS) protocols. Note that the InvestiGATOR end of the cable does not have the usual data set ready (DSR) and ring indicator (RI) inputs. Furthermore, the InvestiGATOR does not offer a protective ground (PGND) input. The protective ground wire from the terminal side of the cable should be connected and provide grounding for the cable shielding. However, the signal ground (SGND) is connected.

## 3.5 I/O Expansion

The I/O expansion connector is intended to allow unforeseen problems to be addressed. The I/O expansion connector is mapped to the to I/O address space and

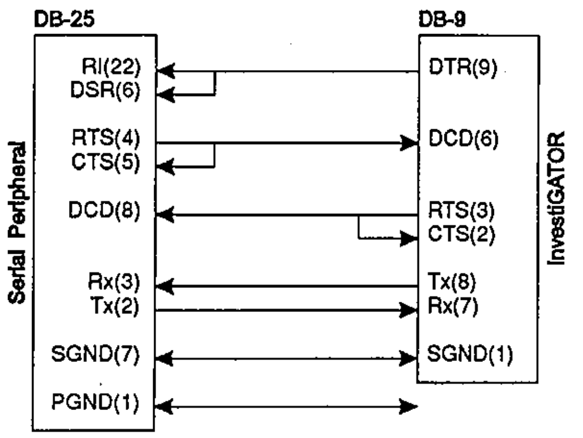


Figure 3.12: InvestiGATOR to IBM PS/2 Serial Cable

may be used with eight, sixteen, and thirty-two bit data bus sizes. Wired-OR lines are provided for asynchronous bus cycle termination and interrupts. The port is fully buffered and the address lines and control lines are always turned on, thus allowing the I/O expansion connector to be used to probe system activity. A list of signal names, pin numbers, and description of the signals' functions are given in Table 3.6.

| Pin Number | Signal Name | Description |
|---|---|---|
| 0—31 | D31-0 | Data bus. |
| 32—51 | A19-0 | Address bus. |
| 52 | AS* | Address strobe. |
| 53 | DS* | Data strobe. |
| 54 | IOSP* | I/O address space flag. |
| 55 | IO8DTACK* | Eight bit port DTACK. |
| 56 | IO16DTACK* | Sixteen bit port DTACK. |
| 57 | IO32DTACK* | Thirty-two bit port DTACK. |
| 58 | IOIRQ* | I/O expansion port IRQ line. |

Table 3.6: I/O Expansion Connector Signals

## 3.6 Array Bus

The array bus is a connection rich environment. Previous experience and analysis has led to the conclusion that interboard connectivity was lacking in traditional host environments such as the PC-XT, PC-AT, EISA, MicroChannel, VME, and others.

The InvestiGATOR has a 324 signal connector. Seventy-five of the signals on this bus are allocated for a memory mapped interface to the MC68030 SCSI control processor. These signals are fixed in terms of arrangement and function. The remaining signals are broken up between near-neighbor connections and broadcast connections which are functionally undedicated *a priori*. One-hundred forty of these signals are wired as near-neighbor connections where seventy of the signals go to the right adjacent slot and the remaining seventy go to the left adjacent slot. The remaining one-hundred nine signals are wired as a broadcast bus to the array. All of the near-neighbor connections are array broadcast connections are invisible to the MC68030 CPU. A breakdown of the allocation of these signals is listed in Table 3.7.

### 3.6.1 CPU to Array Bus Interface and Architecture

The CPU is interfaced to the array bus via a memory mapped interface using a total of seventy-five signal lines on the backplane connector. The interface to the array bus buffers the CPU signals and passes all signals necessary for data and instruction transactions to take place. A breakdown of the allocation of these signals is listed in Table 3.7.

This interface does not support alternate address spaces via the 68030's function code (FCx) outputs, dynamic bus sizing (*i.e.*, all ports are thirty-two bits), nor does it support burst mode accesses. Each slot has its own STERM signal which is routed to the CPU by the interface. STERM validity is ascertained by observation of the SLOTENx signals. Each slot has a wired-OR SLOTEN (active-low) signal which is held high if a card is not present in a slot. If a card is present and needs to be able to assert STERM then it must assert the SLOTENx signal by wiring the signal directly to ground. The STERMx and SLOTENx signals are unique at each connector and are hidden from the other slots.

The array bus error (ARYBERR) and interrupt request (ARYIRQ) signals are wired-OR. ARYBERR causes a BERR cycle to be executed by the 68030, while ARYIRQ requests a level one priority 68030 IRQ.

### 3.6.2 Local (Near-Neighbor) Connections

The local slot connections consist of seventy signal lines to each adjacent slot. While these connections are not predefined, they are adequate to implement a sixty-four bit, bidirectional communication port or a pair of thirty-two bit unidirectional ports to each adjacent slot. These signals are unused in the Gauss machine implementation, but will be used in a future TMS320C40 hypercube implementation.

### 3.6.3 Array Broadcast Bus

The array broadcast bus consists of the remaining 109 signal lines not used in the near-neighbor connections or the CPU-array interface. Like the near-neighbor connections, the broadcast connections are not defined *a priori*. These connections are intended to handle control and data distribution. The assignment of these signals for the Gauss machine is discussed in Section 6.2.

## 3.7 Support Circuitry

This section describes the miscellaneous modules that provide the critical support functions which are not a proper part of any of the major modules of the architecture.

### 3.7.1 Clock Generator Module

The clock generator module consists of three components: the crystal time base, the clock generator, and a low-skew buffer. The crystal timebase is a 40 MHz TTL compatible clock. This clock drives an AMD Am2971A PEG (Programmable Event Generator) which produces phase locked versions of 2 MHz, 5 MHz, 10 MHz, and 20 MHz clocks. Finally, since the PEG has a relatively low power output drive, the clock signals are buffered by an AMD Am29C827A high-speed CMOS bus driver. The Am29C827A features low $t_{PD}$, low skew, and "edge-rate control" which is intended to minimize ground bounce.

The clock module produces one copy each of the 2 MHz and 5 MHz clocks, two copies of the 10 MHz clock, and six copies of the 20 MHz clock. The various copies of the 20 MHz clock are reserved for distribution to different modules, with the intent of minimizing clock skew within each module. The clock distribution reservation table is shown in Table 3.8.

### 3.7.2 Reset Circuit Module

The reset circuit module contains power-up and on demand system reset circuitry. Power-up reset is provided by a Texas Instruments TL7705A Power Supply Supervisor/Reset Generator. The power-up reset circuit monitors system power and asserts the RESET signal for an amount of time controlled by C1. C1 has been chosen to be greater than $40\mu F$, thus, RESET will be asserted for at least 500 ms after the 5V supply rail reaches within ten per cent of 5V.

The reset signal provided by the TL7705A is buffered into the wired-OR system RESET* signal by an open-collector inverter. The reset circuit contains a reset switch connected to the system RESET* signal.

| Pin Number | Signal Name | Description |
|---|---|---|
| 1 | SLOTENx* | Slot enable. Wired-OR. |
| 2 | STERMx* | Synchronous bus cycle termination. |
| 3 | ARYDS* | Data strobe. |
| 4 | ARYAS* | Address strobe. |
| 5 | ARYR/W | Read/write strobe. |
| 6 | ARYUU* | Upper byte select. |
| 7 | ARYUM* | Upper-middle byte select. |
| 8 | ARYLM* | Lower-middle byte select. |
| 9 | ARYLL* | Lower byte select. |
| 10 | ARYARYSP* | Array address space select. |
| 11 | ARYRMC* | Read-modify-write signal. |
| 12 | RESET* | System reset. |
| 13 | HALT* | System halt. |
| 14—45 | D31-0 | Data bus. |
| 46—75 | A29-0 | Address bus. |
| 79 | CLK20C | 20 MHz system clock. |
| 81 | CLK10B | 10 MHz system clock. |
| 83 | CLK5 | 5 MHz system clock. |
| 85 | CLK2 | 2.5 MHz system clock. |
| 77,82,84,87 | Vcc | 5 V power bus. |
| 76,78,80,86 | GND | Ground rail. |
| 88—? | — | Near neighbor connections.<br>Odd pin numbers to left slot.<br>Even pin numbers to right slot. |
| ?—324 | — | Broadcast bus. |

Table 3.7: Array Bus Signals

| Signal | Frequency | Reservation/Availability |
|---|---|---|
| CLK2 | 2 MHz | Unallocated |
| CLK5 | 5 MHz | Unallocated |
| CLK10a | 10 MHz | I/O module |
| CLK10b | 10 MHz | Array module |
| CLK20a | 20 MHz | CPU module |
| CLK20b | 20 MHz | I/O module |
| CLK20c | 20 MHz | Array module |
| CLK20d | 20 MHz | RAM module |
| CLK20e | 20 MHz | ROM module |
| CLK20f | 20 MHz | Unallocated |

Table 3.8: Clock Reservation

# Chapter 4

# SOFTWARE

The InvestiGATOR's firmware is written primarily in C. Besides being readily available for the 68030 architecture, the C language offers high level language benefits of compactness and ease of use combined with some of the benefits associated with assembly language, mainly control and speed. The InvestiGATOR firmware is modular in nature, composed of a kernel, SCSI bus interface (SBIC) firmware, serial I/O (SIO) firmware, and interface code to the target processor, the Gauss machine. A block diagram of the software architecture is shown in Figure 4.1.

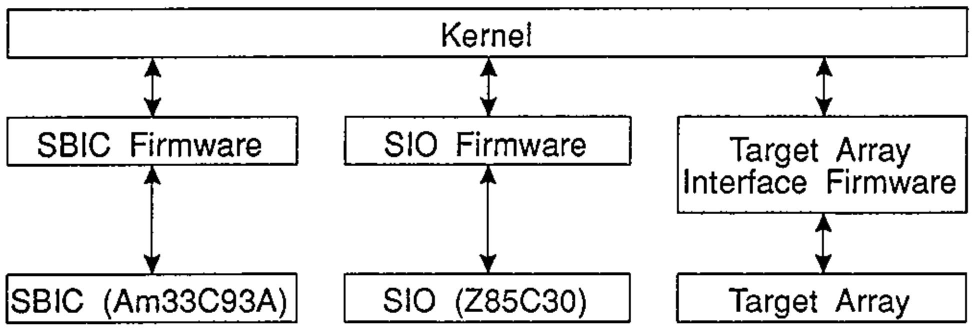


Figure 4.1: InvestiGATOR Software Architecture Block Diagram

## 4.1 Kernel

The primary mission of the kernel is to manage resources and control dispatch of tasks to the various subsystems. The key resource which is managed by the kernel is memory. The kernel also manages the dispatch of interrupts to the various subsystems.

## 4.2 SBIC Firmware

The SBIC firmware is responsible for managing the substantial SCSI protocol. The following sections introduce the operation of the SCSI bus and the structure of the SBIC firmware.

### 4.2.1 SCSI Bus Operation

The SCSI bus has four phases of operation. The SCSI bus idles in the bus free phase. When a device wants to gain control of the bus, the bus enters the arbitration phase. During the arbitration phase all devices attempting to gain control over the bus arbitrate for the bus. The device with the highest SCSI ID wins the arbitration. After successful arbitration the bus enters the selection phase. During the selection phase the SCSI bus master attempts to select the device with which it wants to communicate. After successful selection the bus enters the information transfer phase. The information transfer phase is characterized by the transfer of commands, data packets, and messages. A flow diagram of the SCSI phases is shown below in Figure 4.2.

There are two types of devices on the SCSI bus: initiators and targets. Initiators are typically host processors while targets are typically peripheral devices such as disk drives. The InvestiGATOR operates as a target. The InvestiGATOR responds to the commands test unit ready, request sense, send and receive. These operation of the these commands are shown in Figures 4.3–4.6.

The test unit ready command is used to query the target device as to its status. This command is mandated by the SCSI standard. The InvestiGATOR will

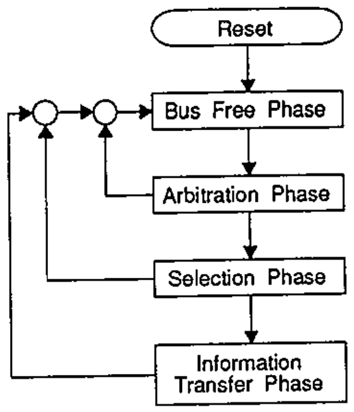


Figure 4.2: SCSI Bus Phases

respond with a good, check condition, or busy status code. The good status code indicates that the InvestiGATOR is ready and standing by for a command. The check condition status code indicates that the InvestiGATOR is not ready and has additional status information available. Finally, the busy status code indicates that the InvestiGATOR is busy. The transactions required to execute a test unit ready command are shown in Figure 4.3.

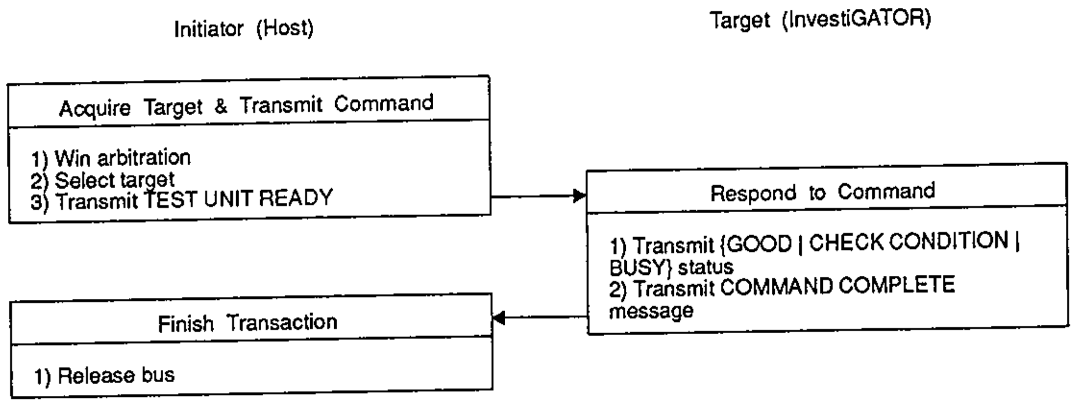

Initiator (Host)
Target (InvestiGATOR)
Acquire Target & Transmit Command
1) Win arbitration
2) Select target
3) Transmit TEST UNIT READY
Respond to Command
1) Transmit {GOOD | CHECK CONDITION | BUSY} status
2) Transmit COMMAND COMPLETE message
Finish Transaction
1) Release bus

Figure 4.3: Test Unit Ready Command Operation

The request sense command is used to query the device for extended status data. Typically, the request sense command is executed after a check condition status is returned on a command. The transaction model for the request sense command is shown in Figure 4.4.

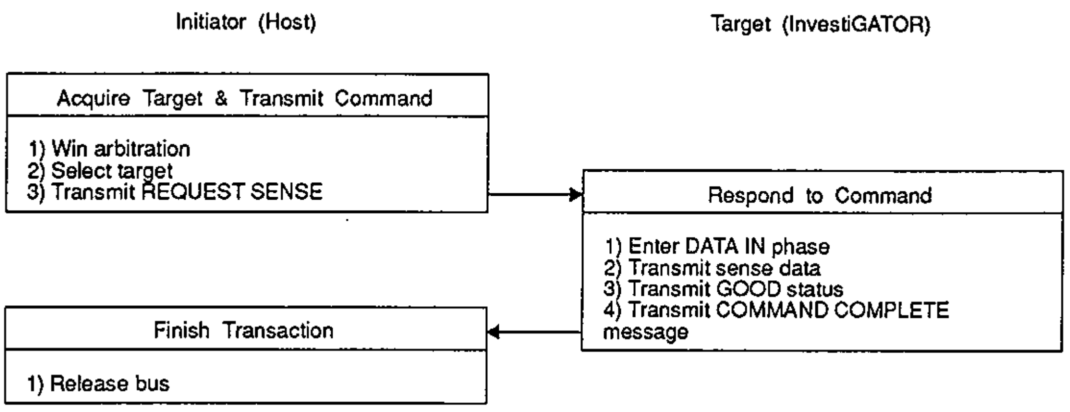


Figure 4.4: Request Sense Command Operation

The send and receive commands are the primary data communication commands between a host processor and the InvestiGATOR. The transaction models for the send and receive commands are shown in Figure 4.5 and Figure 4.6.

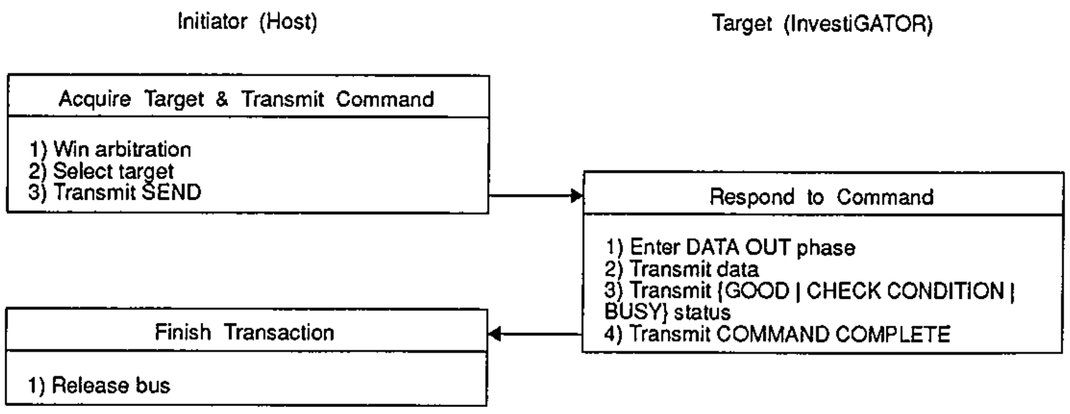


Figure 4.5: Send Command Operation

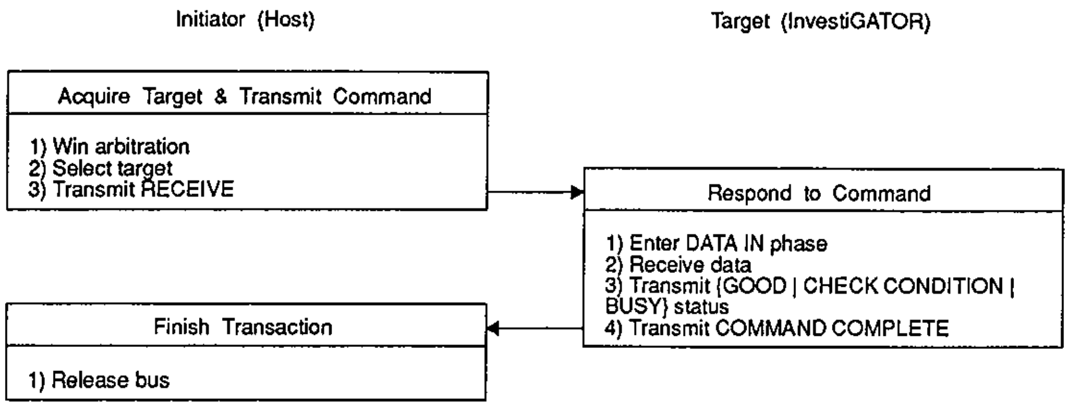


Figure 4.6: Receive Command Operation

### 4.2.2 SBIC Firmware

The SBIC operates under an interrupt driven protocol. This subsection discusses the flow diagram of the SBIC reset routine and the interrupt service routine (ISR) depicted in the flow diagram of Figure 4.7.

Before the SBIC can be used, it must be initialized via a software interrupt. The SBIC is preloaded with the SCSI address of the InvestiGATOR before a software reset is executed. After the reset completes, interrupts and data I/O modes are programmed. Initially, the InvestiGATOR is set to SCSI address 4 and uses interrupt drive I/O.

The InvestiGATOR operates only as a target in the initial configuration. The InvestiGATOR does not support disconnect/reselection at this time so the firmware is fairly simple. The SBIC interrupts the processor with a service required interrupt when an initiator on the SCSI bus selects the InvestiGATOR. Selection may occur either with the attention (ATN) signal asserted or negated: ATN asserted indicates that there is a message pending. Selection with attention is used exclusively to request that the target accept an IDENTIFY message. The InvestiGATOR does not currently support the IDENTIFY message, and thus selection with attention leads to a fault condition.

After the SBIC ISR identifies a selection without attention condition, the ISR prepares the SBIC to receive a command from the initiator. Currently the InvestiGATOR only supports a SCSI command set which is (coincidently) limitted to those commands which have six byte command frames. Thus, a transfer count of six is loaded into the transfer count register and a RECEIVE COMMAND command is issued to the SBIC. The SBIC then receives a command from the initiator.

If a data phase is required by the command received from the initiator then the SBIC is prepared for a data phase by setting the synchronous transfer control register and the transfer counter register and issuing a send or receive data command.

If the command received was a linked command then a SEND STATUS command is issued to the SBIC and the execution returns to the RECEIVE COMMAND phase. If the command was not a linked command then a SEND STATUS AND COMMAND COMPLETE command is issued to the SBIC, causing the last command's status to be transmitted to the initiator and the SBIC to disconnect.

## 4.3 SIO Firmware

The serial port is operated in an interrupt driven I/O mode. The SIO drivers support circular transmit/receive buffers which aid in increasing system throughput and allowing type-ahead. The XON/XOFF flow control protocol is the only flow control protocol currently supported. In the current implementation serial port A is the console (stdin/stdout) while serial port B is unassigned.

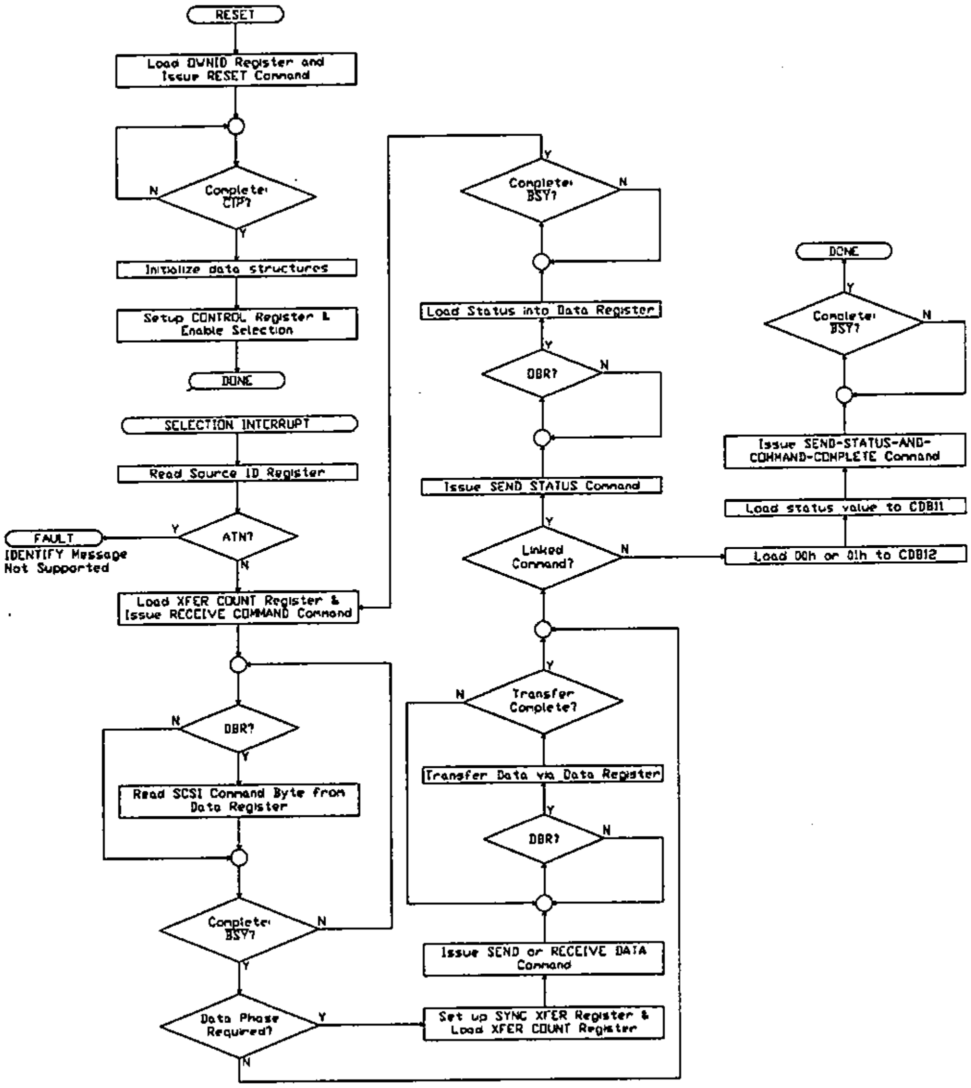

RESET
Load OWNID Register and Issue RESET Command
Complete CIP?
N
Y
Initialize data structures
Setup CONTROL Register & Enable Selection
DONE
SELECTION INTERRUPT
Read Source ID Register
ATN?
Y
N
FAULT
IDENTIFY Message Not Supported
Load XFER COUNT Register & Issue RECEIVE COMMAND Command
DBR?
N
Y
Read SCSI Command Byte from Data Register
Complete BSY?
N
Y
Data Phase Required?
Y
N
Complete BSY?
Y
N
Load Status into Data Register
DBR?
Y
N
Issue SEND STATUS Command
Linked Command?
Y
N
Transfer Complete?
Y
N
Transfer Data via Data Register
DBR?
Y
N
Issue SEND or RECEIVE DATA Command
Set up SYNC XFER Register & Load XFER COUNT Register
DONE
Complete BSY?
Y
N
Issue SEND-STATUS-AND-COMMAND-COMPLETE Command
Load status value to CDB11
Load 00h or 01h to CDB12

Figure 4.7: SBIC Interrupt Service Routine Flow Diagram

# Part III

# Gauss Machine

Chapter 5

# INTRODUCTION

The Gauss machine is a $2 \times 2$ systolic array processor comprised of three seven-bit GEQRNS channels for a total of six seven-bit RNS channels. The array of processors is arranged in a mesh-connected topology with unidirectional dataflow. Alternately, the Gauss machine may be configured to utilize two of its processors as a vector processor. The Gauss machine excels in computation of level 3, level 2, and level 1 operations.

## 5.1 Motivation

The design of the Gauss machine is motivated by several factors. There exists a need for high-performance front-end signal processors which are reliable, small, consume minimal power, and are relatively inexpensive. Typically, high performance is achieved using a combination of fast processors coupled with some parallelism. Signal processing applications have been demonstrated to be particularly amenable to systolic array implementations[7]. Traditional technologies have typically featured large, multiple package designs where individual processors were made up of several large VLSI devices[8]. Even new, state-of-the-art processors designed for parallel processing, but based on conventional arithmetic technology such as the iWarp[9] or TMS320C40[10] have at least one large package per processor element. These designs typically had large physical form factors, high power consumption (multiple watts per processor), and low reliability. Attempts to improve reliability by incorporating redundancy typically result in little improvement at the expense of greater than one-hundred per cent in terms of hardware, power, size, and cost.

Processor architectures based upon residue arithmetic are uniquely qualified to meet the demanding needs of modern signal processing systems. The RNS is a high performance system of arithmetic having performance which is independent of word-width. The RNS features relatively small die area when compared with conventional arithmetic. The RNS is inherently fault and defect tolerant[3, 4], and may realize the full potential of VLSI systolic arrays[7].

## 5.2 Design Parameters

Currently, there are no RNS systems which are general purpose in nature. Most RNS systems are hard-wired to a specific task. There exists a need to demonstrate an RNS system which is more general purpose in nature. This RNS system must be capable of many different operations. Additionally, there is motivation to demonstrate the use of the RNS in systolic array architectures.

The Gauss machine is designed as a discrete prototype of a $2 \times 2 \times 6$ VLSI systolic array of GEQRNS multiplier-accumulators. The array is hosted by the InvestiGATOR array processor testbed. Data conversion functions are provided by the InvestiGATOR. The array controller is a microprogrammed controller based upon a single chip microsequencer.

# Chapter 6

# IMPLEMENTATION

## 6.1 Architecture

The Gauss machine supports a three channel GEQRNS or QRNS, $2 \times 2$ array of seven bit multiplier-accumulators. The array is formed by six boards, with each board comprising a $2 \times 2$ array seven bit multiplier-accumulators. The array is integrated into the InvestiGATOR array processor backplane with the addition of a controller, and optionally, a forward-conversion and CRT engine board.

The Gauss machine supports a mesh connected geometry with north and east flow of data. The array uses FIFOs to provide the means for data to be sequenced through the array. The FIFOs are the gateway through which the array communicates with the outside world. Additionally, the Gauss machine offers a vector mode of operation which utilizes PEs (1,1) and (1,2) to perform level 1 and level 2 operations at higher performance levels than would be possible using the full array. A block diagram is given below in Figure 6.1.

The FIFOs located on the periphery of the array meet the goal of allowing concurrency in processing and data I/O since the memories may be loaded or emptied as calculations proceed.

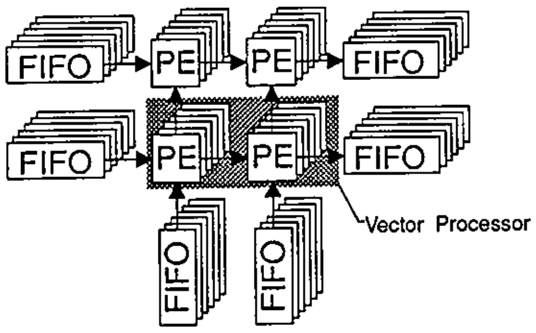


Figure 6.1: Block Diagram of Gauss Machine Array

## 6.2 Processor Implementation

Each processor element in the array (see Figure 6.1) consists of a multiplier, accumulator, and support architecture. The inputs to the multiplier come from the X-bus and Y-bus. The X-bus is also connected to the F-bus, allowing the accumulator to be pre-loaded, or the output of the adder may be output to the X-bus. A block diagram of the processor element is depicted below in Figure 6.2.

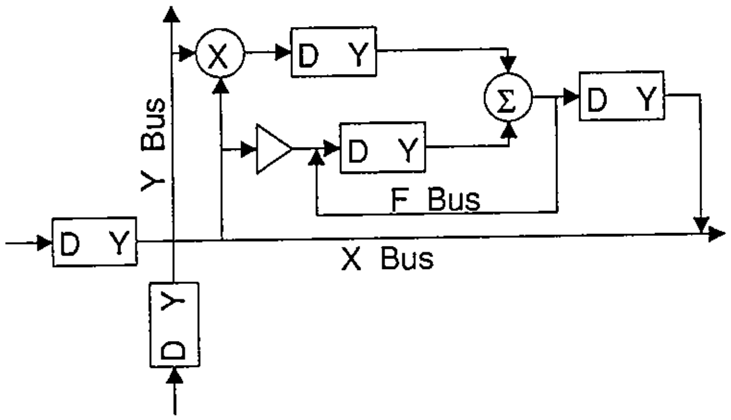


Figure 6.2: Block Diagram of Gauss Machine Processor Element

The arithmetic units in this discrete implementation are direct lookup tables implemented in static RAM. In a VLSI implementation these arithmetic units would be implemented with adders and small ROM lookup tables. Additional architectural enhancements are made to PEs (1,1), and (1,2) to allow these two processors to operate as a very high throughput vector processor. The array architecture in vector mode is shown in Figure 6.3. The augmented processor is depicted in Figure 6.4.

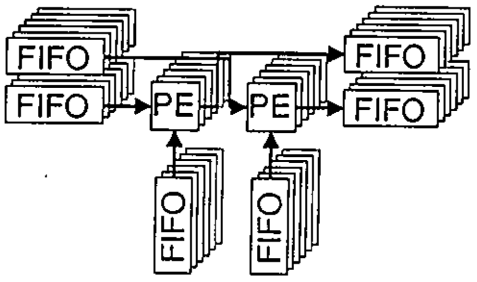


Figure 6.3: Block Diagram of Vector Mode Architecture

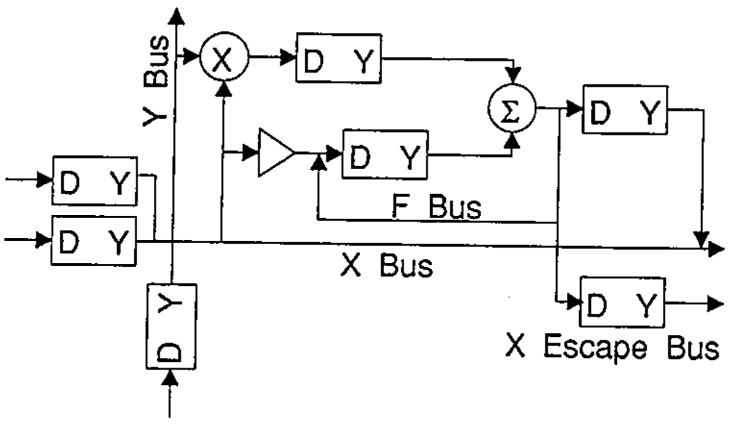


Figure 6.4: Augmented Processor Element

The X-bypass-bus of the enhanced PE is connected to the X-FIFOs, allowing two operands per cycle to be deposited on each of the enhanced processors. The X-escape bus of PE (1,1) allows the results to be flushed out of the processors in one clock cycle. The vector enhancement allows the Gauss machine to perform level 1 and level 2 operations very efficiently, and while the enhancement does not allow an addition of two operands to be performed directly, it may be performed in two cycles using the accumulator. The vector processor can also perform pointwise multiplication of two vectors using a single clock cycle per operand pair.

### 6.2.1 Processor Control Signals

This section lists the processor control signals and their function. The control signals are registered on the processor boards. The signals are listed in Table 6.1.

The signals in Table 6.1 may be broken into several groups. These groups are:

- Address information: BA2-0, and PA2-0.
- FIFO Control: XIW*, YIW*, XOW*, XIR*, YIR*, XOR*, XIFLRT*, YIFLRT*, and XOFLRT*.
- Adder RAM Control: ROE*, ARWE*.
- Multiplier RAM Control: MROE*, MRWE*.
- X-Bus Control: XBOE*, XBEN*, XFEN*, AREN*, and AROE*.
- Y-Bus Control: YBEN*.
- Processor Structure Control: PREN*, and SREN*.
- Processor Configuration Control: VECTORMODE and ARITHMODE.
- Miscellaneous: CLR*, RESBWE*, and RESBRE*.

## 6.3 Controller Implementation

The Gauss machine uses a microprogrammable controller. The heart of the controller is a single chip microsequencer with EPROM based microprogram store, the AMD Am29CPL154. The microcode store has a total of 512 words of microinstruction storage. The microsequencer uses PLDs to decode its instructions for the array. The architecture of the controller is depicted in Figure 6.5. The Gauss machine controller has a pipeline delay model depicted in Figure 6.6.

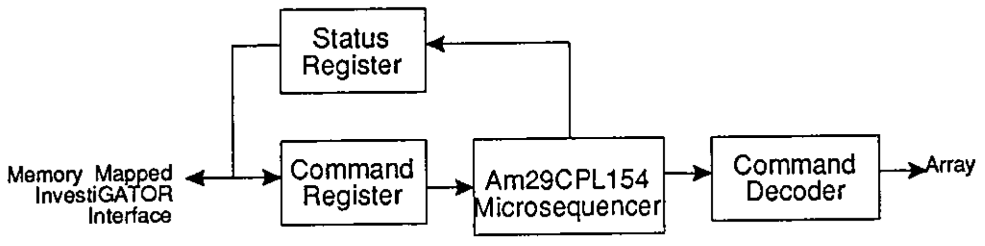

Status Register
Memory Mapped InvestiGATOR Interface
Command Register
Am29CPL154 Microsequencer
Command Decoder
Array

Figure 6.5: Block Diagram of Gauss Machine Controller Architecture

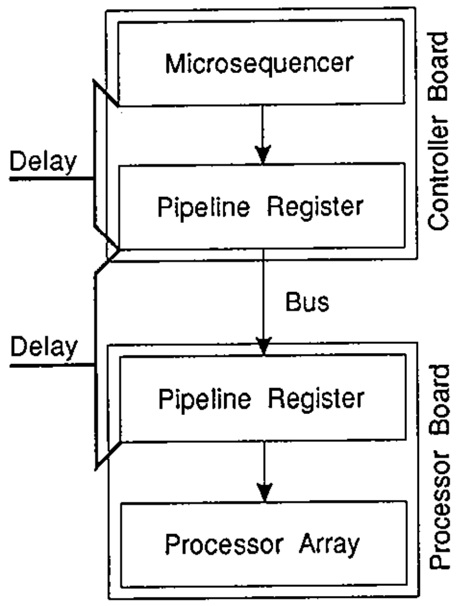


Figure 6.6: Gauss Machine Pipeline Delay Model

In order to perform an operation on the array, the InvestiGATOR will load some data into the array input FIFOs, and order the controller to perform the expected operation by writing a command to the command register. The InvestiGATOR then monitors the status register in order to determine when the computation is complete. Then the InvestiGATOR retrieves the results from the array output FIFOs.

This same method is used for programming the array multipliers and adders, except that there is no need to read back any results.

The chosen microsequencer, the Am29CPL154 has a relatively narrow output word (eight bits), yet the array has a substantial number of control lines as evidenced by Table 6.1. Fortunately, this does not present a problem because there are only a limited number of useful combinations of control signals. Therefore, the output word of the microsequencer is used as a command code or instruction and is decoded into the appropriate set of signals by the command decoder, see Figure 6.5.

## 6.4 Array Initialization

In order to perform useful operations on the array, the arithmetic elements must be initialized. There exist enhancements which are not visible in the block diagram of Figure 6.2 to allow programming of the multiplier. The adder can be programmed without any architectural enhancements.

The multiplier and architecture related to its programming is depicted in Figure 6.7. Control signals are indicated in the block diagram. The multiplier memory is addressed by the X-bus and Y-bus, and by the ARITHMODE signal. The multiplier data is loaded from the X-bus to the multiplier memory. Register output enable signals are indicated by an OE suffix while latch enable signals are indicated by an EN suffix. The write strobe for the memory is indicated by the MRWE* signal. The MRWE* signal is broadcast to all processors in the system so all of the multipliers must be programmed at the same time.

Programming of the multiplier proceeds as follows. The X- and Y- FIFOs are loaded by the InvestiGATOR. The InvestiGATOR sends a command to the Gauss machine controller to program a block of the multiplier memory. X- input FIFO

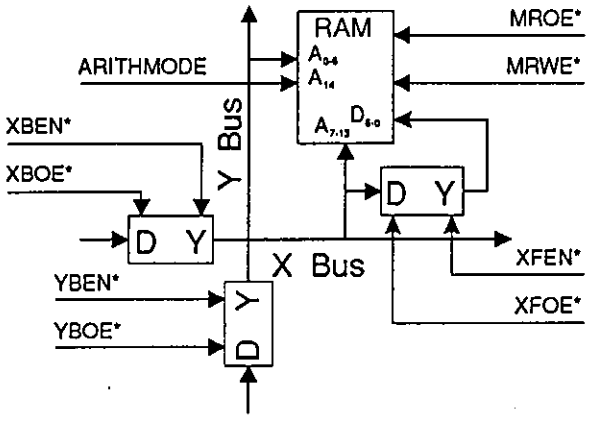


Figure 6.7: Processor Multiplier Programming Model

transmits the contents of the memory location across the X- bus to a register which outputs to the multiplier memory's data bus. Next, the address of the data word to be programmed is propagated across the array from the X- and Y- input FIFOs and the multiplier memory's write line is strobed. The process is repeated until the multiplier is programmed.

The adder and architecture related to its programming is depicted in Figure 6.8. The adder is programmed as follows. The adder data and addresses are loaded into the X- input FIFO. The least significant portion of the address is transmitted via the X-bus to the product output registers, controlled by PROE* and PREN*, with MROE* negated. Next, the most significant word of address is transmitted and loaded into the accumulator register, controlled by SROE* and SREN*. Finally, The actual data word is transmitted via the X-bus to the F-bus by way of the buffer controlled by XPOE*, and to the adder memory's data port. The adder memory write signal, ARWE*, is strobed, loading the data into the adder memory. This process is repeated until the adder is programmed.

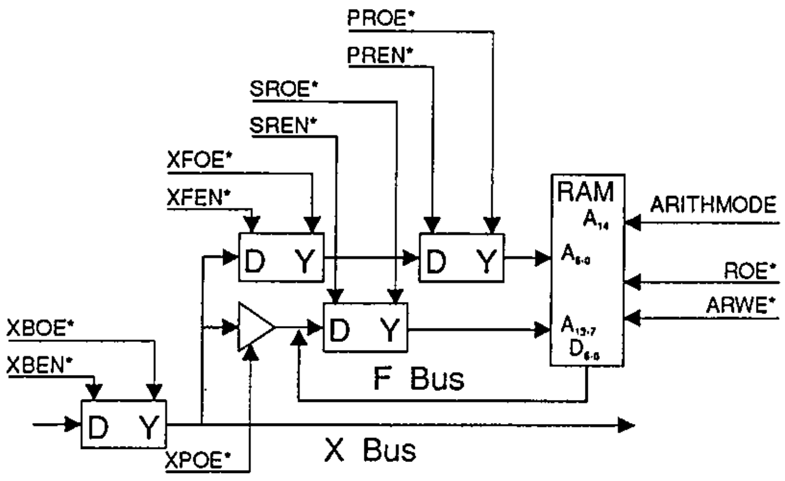


Figure 6.8: Processor Adder Programming Model

## 6.5 Conversion Engine Architecture

The forward conversion engine performs the task of generating the residues of the value input to the engine. This forward conversion is a relatively straightforward process once it is seen that the process may be accomplished simply by breaking the input values into a set of partial sums where each sum represents a range of bits of that number; in other words, suppose we wish to compute the residue modulo $p$ of an $L$ bit number $N$. We would note that the following congruence holds:

$$< N >_p \equiv \left( \sum_{i=0}^{L-1} a_i 2^i \right) \pmod{p},$$

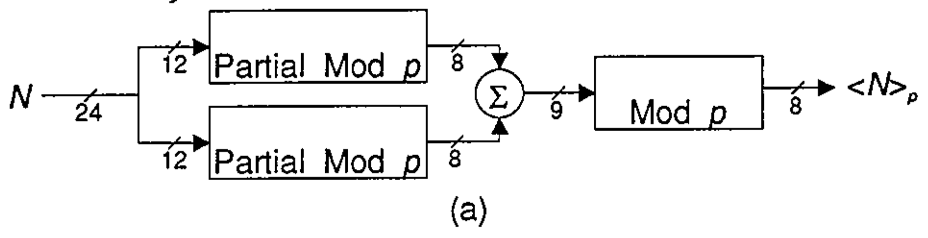
where $a_i \in \{0, 1\}$ and are digits of the binary representation of $N$. Now, suppose $0 < J < K < L - 1$. Then

$$< N >_p \equiv \left[ \sum_{i=0}^{J-1} a_i 2^i + \sum_{i=J}^{K-1} a_i 2^i + \sum_{i=K}^{L-1} a_i 2^i \right] \pmod{p}.$$

A (mod $p$) operation may be added after each partial sum without changing the result:

$$< N >_p \equiv \left[ \left( \sum_{i=0}^{J-1} a_i 2^i \right) (\text{mod}p) + \left( \sum_{i=J}^{K-1} a_i 2^i \right) (\text{mod}p) + \left( \sum_{i=K}^{L-1} a_i 2^i \right) (\text{mod}p) \right] (\text{mod}p).$$

This suggests that each partial sum, modulo $p$, can be computed using a small table, and the partial sums added together to form a sum which must be corrected modulo $p$. This is illustrated in Figure 1.2. In Figure 6.9a, conversion of a twenty-four bit input using two tables of order $2^{12}$ to produce an eight bit output is demonstrated. In Figure 6.9b, the same conversion is accomplished using three tables of order $2^8$.



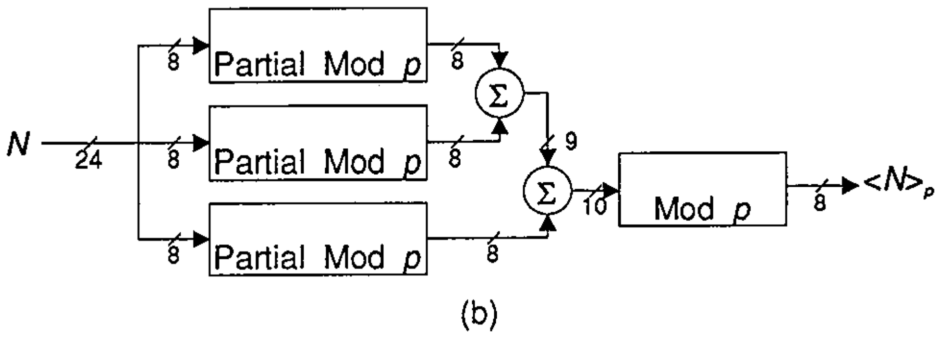


Figure 6.9: Forward Conversion Architecture

The forward conversion engine was not implemented in hardware since it would be relatively expensive to produce a discrete implementation. Instead, the forward conversion engine was implemented with a software architecture inspired by the above discussion. This was motivated by the low speed of a direct implementation of the forward conversion using the standard sequence of divide, multiply, and subtract operations. In particular, the multiplication and division operations are particularly time consuming on the MC68030 (and most microprocessors). The source code in Section D.5 of Appendix D implements a high speed forward conversion based upon table lookup using small tables and minimal arithmetic (addition and subtraction only).

Similarly, the CRT engine hardware was too expensive to implement; emulation of the CRT was substituted. As for the forward conversion, the QRNS to CRNS to Gaussian integer conversion was implemented using a fast, table lookup based algorithm based upon the discussion in Section 1.3. The source code for this high performance implementation is included in Section D.5 of Appendix D.

## 6.6 Application Programmer's Interface

### 6.6.1 Overview

The system software for the Gauss Machine is divided into two parts: firmware for the backplane and the Application Programmer's Interface (API). This chapter describes the API which contains routines for linear algebra and communication between the host and the Gauss Machine. The API is written in THINK C 5.0 for the Macintosh.

The Application Programmer's Interface (API) contains roughly X subroutines that facilitates programming of the Gauss Machine. The idea behind the API is to provide fast prototyping environment for developing and testing new algorithms for the Gauss Machine. Therefore, the routines are not necessarily optimized for speed.

The API can be divided into "high-level" and "low-level" calls. The high-level routines often mimic Matlab statements, e.g., matrix-matrix, matrix-vector, vector-vector multiplication is handled by one routine called `mult()`. The low-level calls implements the primitive operations from which the high-level routines are composed of, such as, memory management and communication between the host and the Gauss Machine. Furthermore, the algebra routines comes in two versions, one using floating-point arithmetic and the other using integer arithmetic.

Typically, the development of an algorithm for the Gauss Machine consists of the following steps:

- Program and test algorithm in Matlab.
- Port Matlab code into API calls.
- Test API code with the Gauss Machine.
- If optimization is of interest, rewrite code using the low-level API.

A complete listing of the API calls are found in Appendix X.

### 6.6.2 High-Level API Routines

Prototyping and testing signal processing/linear algebra algorithms are easily done in interactive packages like Matlab, Mathematica, Maple and Monarch/Siglab. The design of the high-level API was done with this in mind. The API routines imitates Matlab function calls which makes it easy to port an m-file or a Matlab script to a C program running on the Gauss Machine. The Matlab statements are simply exchanged to the corresponding API calls and, with some glue code, the port is complete.

The software was written in THINK C version 5.0 with the following libraries: ANSI, MacTraps. The code was compiled and run on a Mac IIx, 4Mb RAM, 4Mb virtual memory, System 7.0.

These are the THINK C settings under `Edit, Options...`

- **Language Settings**
  - ANSI Conformance
  - Check pointer types.
  - Language Extensions
  - THINK C
  - Strict Prototype Enforcements
  - Infer Prototypes
- **Compiler Settings**
  - Generate 68020 instructions
  - Generate 68881 instructions
  - Classes are indirect by default
  - Methods are virtual by default
  - Optimize monomorphic methods
  - `\bslash p` is unsigned char[]
- **Code Optimization**
  - Defer & combine stack adjust
  - Suppress redundant loads
  - Automatic Register Assignment `Debugging`

- Use source debugger
- Use second screen
- Always save session

These are the THINK C settings under `Project, Set Project Type...`

- Application
- File type APPL
- Partion (K) 384
- Size Flags 0000

The software consists of 5 "library" files (with corresponding header files) and one global header file:

`types.h`: Global type definitions.

`list.c`: Memory management routines. This software was originally written by R. F. Starr, 2639 Valley Field Dr., SugarLand, TX 77479 and was published in Dr. Dobbs Journal. `list.c` have been slightly modified.

`utils.c`: Utilities.

`conv.c`: Floating-point to fixed-point conversion routines.

`matrix.c`: Floating-point matrix algebra routines, memory management.

`int_matrix.c`: Integer matrix algebra routines, memory management.

The routines in `int_matrix.c` are identical to the routines in `matrix.c` except for those operations that are not defined for integers, i.e., division.

In order to compile these files as parts of a code resource, change all calls of `malloc()` to `NewPtr()` and `free()` to `DisposPrt()`. Furthermore, comment out the `stdio` routines, i.e., `printf()` and friends, in `utils.c`. It may be also necessary to change the ANSI library to the required library for code resources.

Note: Whenever the comments in the code and this document disagree, rely on this document.

### 6.6.3 Macros and Constants

file: `int_matrix.c`

```
#define COMP 0x4 /* marks compatible dimensions */

#define SCAL 0x8 /* marks one operand as a scalar */

#define INT(a) ((int)(a)) /* casts a to integer */

#define EQDIM(a, b) ( (a->rows == b->rows) && (a->cols ==
b->cols) ) /* checks if a and b has the same dimensions */
```

file: `matrix.c`

```
#define OOPS printf(oops: %d\n, __LINE__); /* debugging macro */

#define COMP 0x4 /* marks compatible dimesions */

#define SCAL 0x8 /* marks one operand as a scalar */

#define INT(a) ((int)(a)) /* casts a to integer */
```

file: `conv.h`

```
#define max(a, b) (a > b) ?  a :  b /* maximum of a and b */

#define min(a, b) (a < b) ?  a :  b /* minimum of a and b */
```

file: int_matrix.h]

```
#define deref(type,x) *((type*)(x)) /* not really useful */
```

file: matrix.h

```
#define SIZE(a) ((a)->rows * (a)->cols) /* computes number of
elements in matrix */

#define EQDIM(a, b) ( (a->rows == b->rows) && (a->cols ==
b->cols) ) /* checks if a and b has the same dimensions */

#define cmul(a, b, c, d, e, f);  a = (c) * (e) - (d) * (f); b =
(c) * (f) + (d) * (e);  /* complex multiply */

#define cabs(a, b) sqrt(((a) * (a) + (b) * (b))) /* compute
complex absolut value */
```

file: types.h

```
#define INTTYPE long /* integer data type */

#define FLOATTYPE double /* floating-point data type */

#define NOERR 0 /* OK return code */

#define CMPLX 0x1 /* marks a complex value */

#define REAL 0x2 /* marks a real value */
```

file: utils_h

```
#define PLAIN 0x0 /* Plain format */

#define MATLAB 0x1 /* Print in MATLAB style (with [ ] and ;) */
```

### 6.6.4 Function Descriptions.

matrix *add(matrix *a, matrix *b)

description: Adds matrices a and b.

arguments: `matrix *a, *b` Input matrices.

returns: `matrix *` The sum of a and b, NULL if error.

usage:
```
sum = add(a, b); /* sum = a + b

matlab equivalent:

>> sum = a + b; */
```

file: `matrix.c`

matrix *appendcols(matrix *a, matrix *b)

description: Returns a matrix with b's columns appended to a ([a, b]). Naturally, a and b must have the same number of rows.

arguments: `matrix *a, *b` Input matrices.

returns: `matrix *` [a, b], NULL if error.

usage:
```
c = appendcols(a, b); /* c = [a, b]

matlab equivalent:
```

```
>> c = [a, b]; */
```

file: `matrix.c`

**matrix *appendrows(matrix *a, matrix *b)**

description: Returns a matrix with b's rows appended to a ([a; b]). Naturally, a and b must have the same number of columns.

arguments: `matrix *a, *b` Input matrices.

returns: `matrix *` [a; b], NULL if error.

usage:
```
c = appendrows(a, b); /* c = [a; b]

matlab equivalent:

>> c = [a; b]; */
```

file: `matrix.c`

**matrix *assign(matrix *target, matrix *rows, matrix *cols, matrix* *source)**

description: Puts the matrix source into a sub matrix of target indicated by rows and cols. That is, rows and cols defines a sub matrix of target (exactly like sub_matrix()) and this sub matrix is overwritten with data from the source matrix. This is analogous to the matlab statement target(rows, cols) = source. Needless to say, the sub matrix of target and source must be of the same dimensions. For example, suppose

target = [1 2 3 4; 5 6 7 8; 9 10 11 12], source = [13 14; 15 16],

rows = [3 2] and cols = [1 2], the resulting matrix would be

[ 1 2 3 4; 15 16 7 8; 13 14 11 12 ].

If rows or cols is NULL, this means all the rows and all the columns of target. That is, target(rows,:) = source would be coded as assign(target, rows, NULL, source), and similarly, target(rows,:) = source would be coded as assign(target, rows, NULL, source).

arguments: `matrix *target` Matrix to be written to.

`matrix *rows` Row indexing matrix.

`matrix *cols` Column indexing matrix

`matrix *source` Matrix whose data will be written to target.

returns: `matrix *` Copy of target with parts overwritten by source, NULL if error.

usage:
```
assign(target, rows, cols, source); // target(rows, cols) =
source
assign(target, rows, NULL, source); // target(rows, :)  = source
assign(target, NULL, cols, source); // target(:, cols) = source
```

see also: `sub_matrix() temp_copy()`

note: Does not handle the case target(:,:) = source. For this case copy source with target = temp_copy(source).

file: `matrix.c`

## void clear_error(void)

description: Clears the error string. This is typically done at start up or after recovering from an error.

arguments: nothing

returns: nothing

usage: `clear_error(); // Clear error messages`

see also: `get_error(), error(), print_error()`

file: `utils.c`

## void close_GM(void)

description: Frees memory allocated to temporary matrices and cleans up. close_GM() should only be called once and be matched with a open_GM() call. To only free memory allocated by temporary matrices use kill_temp_list().

arguments: none

returns: nothing

usage: `close_GM(); // Clean up`

see also: `open_GM(), kill_temp_list()`

file: `utils.c`

## int cmplx_promote(matrix *a, matrix *b)

description: Promotes, if necessary, the operands a and b to complex matrices, that is, if either a or b is complex then both a and b are converted to complex matrices.

arguments: `matrix *a, *b` Matrices to be promoted.

returns: `int` NOERR if successful, −1 if malloc failed

usage: `res = cmplx_promote(a, b); /* Complex promotes a and b, OK if res == NOERR. */`

see also: `op_check()`

file: `matrix.c`

## matrix *conj(matrix *mat)

description: Returns a matrix equal to the complex conjugate of the input matrix.

arguments: `matrix *mat` Input matrix.

returns: `matrix *` The complex conjugated input matrix

usage:
```
conj_A = conj(A); /* conj_A = -A

matlab equivalent:

>> conj_A = conj(A); */
```

file: `matrix.c`

## matrix *copy_matrix(matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with new_matrix(). Note that it is the user's responsibility to free this matrix. Use copy_temp() to get a temporary matrix (which will be freed by kill_temp_list() or close_GM()).

arguments: `matrix *source` Matrix to be copied.

returns: `matrix *` Copy of source. NULL if out of memory.

usage: `new = copy_matrix(old); // copy the matrix old to the matrix new`

see also: `kill_matrix(), new_temp(), copy_temp(), new_matrix(), kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with copy_matrix (with kill_matrix).

file: `matrix.c`

## matrix *copy_temp(matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with new_temp() and therefore, is a temporary matrix. To free ALL temporary matrices, use kill_temp_list() or close_GM().

arguments: `matrix *source` Matrix to be copied.

returns: `matrix *` Copy of source. NULL if out of memory.

usage: `new = copy_temp(old); // copy the matrix old to the temporary matrix new`

see also: `kill_matrix(), copy_matrix(), new_temp(), new_matrix(), kill_temp_list()`

file: `matrix.c`

## void error(char *msg)

description: Copies the string msg to the global error string. This routine is used to report errors. The error string can be recovered by get_error(). Maximum length of msg is 255 characters.

arguments: `char *msg;` Error message to be copied to the global error string.

returns: nothing

usage: `error(Division by zero is a bad idea); /* Division by zero error message */`

see also: `get_error(), clear_error(), print_error()`

file: `utils.c`

## FLOATTYPE fixed2float(INTTYPE i)

description: Converts a fixed-point number to floating-point using the word length and number of fractional bits set by init_conv().

arguments: `INTTYPE i` Fixed-point number to be converted to floating-point.

returns: `FLOATTYPE` Floating-point number representing the input argument.

usage:
```
FLOATTYPE result;
INTTYPE int_result;
result = float2fixed(int_result); /* convert int_result to
floating-point */
```

see also: `init_conv(), float2fixed(), mfloat2fixed(), mfixed2float()`

file: `conv.c`

### INTTYPE float2fixed(FLOATTYPE f)

description: Converts a floating-point number to fixed-point using the word length and number of fractional bits set by init_conv().

arguments: `FLOATTYPE f` Floating-point number to be converted to fixed-point.

returns: `INTTYPE` Integer whose bits are the fixed-point representation of the input argument.

usage:
```
INTTYPE fixed_pi;
fixed_pi = float2fixed(3.141592654); // convert pi to fixed-point
```

see also: `init_conv(), fixed2float(), mfloat2fixed(), mfixed2float()`

file: `conv.c`

## void get_error(char *error_string)

description: Copies the global error string to error_string. If an error has occurred the error string will contain an error message. error_string must be allocated to at least 255 characters by the caller.

arguments: `char *error_string` Will contain a copy of the error message (if any).

returns: nothing

usage:
```
char err_str[255]; // make sure that err_str is at least 255
chars long
get_error(err_str); // get error message
```

see also: `error(), clear_error(), print_error()`

file: `utils.c`

## void GM2LV(matrix *a, TD1Hdl re, TD1Hdl im)

description: Copies data from GM matrix to Labview matrix. Note that re and im must be already allocated by the caller and of correct dimensions. If a is a real matrix then zeros will be put in im.

arguments: `matrix *a` GM matrix whose data is to be copied to re and im.

`TD1Hdl re, im` Handles to Labview matrix data structure.

returns: nothing

usage: `LV2GM(A, A_real, A_imag); // Copies data from A into A_real and A_imag.`

see also: `LV2GM()`

file: `utils.c`

### matrix *herm(matrix *mat)

description: Returns the conjugate transpose of the input matrix , that is, takes the hermitian of mat.

arguments: `matrix *mat` Input matrix.

returns: `matrix *` The conjugate transposed input matrix, NULL if error.

usage:
```
tran_A = herm(A); /* tran_A = A'

matlab equivalent:

>> A = A'; % Note ' not .', that is, conjugate transpose */
```

see also: `transp()`

file: `matrix.c`

### matrix *imag(matrix *mat)

description: Returns a matrix containing the imaginary part of mat.

arguments: `matrix *mat` Input matrix.

returns: `matrix *` Imaginary part of mat.

usage:
```
im_part = imag(cmplx_matrix); /* im_part = Im[cmplx_matrix]
matlab equivalent:
>> im_part = imag(cmplx_matrix); */
```

file: `matrix.c`

## matrix *index_cols(matrix *mat, matrix *ind)

description: Returns the columns of mat that are pointed out by ind. The elements of ind are truncated to integers (with mfloor()) and are used to pick out columns. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3.14 1.99], then the result would be a matrix of the form [3 2; 6 5; 9 8]. This is analogous to the matlab statement, mat(:, ind).

arguments: `matrix *mat` Matrix to be indexed.

`matrix *ind` Column indexing matrix.

returns: `matrix *` The indexed input matrix, NULL if error.

usage:
```
B = index_cols(mat, ind); /* B = mat(:, ind)
matlab equivalent:
>> B = mat(:, ind); */
```

see also: `index_rows(), index_rows_cols(), sub_matrix()`

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: `matrix.c`

## matrix *index_rows(matrix *mat, matrix *ind)

description: Returns the rows of mat that are pointed out by ind. The elements of ind are truncated to integers (with mfloor()) and are used to pick out rows. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3.14 1.99], then the result would be a matrix of the form [7 8 9; 4 5 6]. This is analogous to the matlab statement, mat(ind, :).

| | | |
|---|---|---|
| arguments: | `matrix *mat` | Matrix to be indexed. |
| | `matrix *ind` | Row indexing matrix. |
| returns: | `matrix *` | The indexed input matrix, NULL if error. |

usage:
```
B = index_rows(mat, ind); /* B = mat(ind, :)

matlab equivalent:

>> B = mat(ind, :)  */
```

see also: `index_cols(), index_rows_cols(), sub_matrix()`

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: `matrix.c`

## matrix *index_rows_cols(matrix *mat, matrix *ind_a, matrix *ind_b)

description: Returns the rows and columns of mat that are pointed out by ind_a and ind_b. The elements of ind_a and ind_b are truncated to integers (with mfloor()) and are used to pick out rows and columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind_a = [3.14 1.99] and ind_b = [2.1], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(ind_a, ind_b).

arguments: `matrix *mat` Matrix to be indexed.

`matrix *ind_a` Row indexing matrix.

`matrix *ind_b` Column indexing matrix.

returns: `matrix *` The indexed input matrix, NULL if error.

usage:
```
B = index_rows_cols(mat, ind_a, ind_b); /* B = mat(ind_a, ind_b)

matlab equivalent:

>> B = mat(ind_a, ind_b); */
```

see also: `index_rows(), index_cols(), sub_matrix()`

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: `matrix.c`

## void init_GM(void)

description: Initializes everything. Clears the global error string (see error()), sets output print format to MATLAB and 8 digits (see init_print()) and initializes memory management (see init_temp_list()). init_GM() should only be called once and be matched with a close_GM() call.

arguments: none

returns: nothing

usage: `init_GM(); // Initialize everything`

see also: `close_GM()`

file: `utils.c`

**void init_conv(int wlen, int fbits)**

description: Initializes the conversion routines and sets the word length and number of fractional bits for the fixed-point representation. The fixed-point numbers are assumed to be signed. Thus, for a word length of 8 and 3 fractional bits, one bit would be the sign bit and 4 bits will be left for the integer part. This means that number between 1111.111 and -1111.111 ($\pm 15.875$) can be represented. Maximal word length is 32.

| | | |
|---|---|---|
| arguments: | `int wlen` | Number of bits per word |
| | `int fbits` | Number of fractional bits |

returns: nothing

usage:
```
init_conv(21, 6); /* Use 21 bit signed words; 6 fractional bits,
14 integer bits */
```

see also: `float2fixed(), fixed2float(), mfloat2fixed(), mfixed2float()`

file: `conv.c`

**void init_print(int sdig, int format)**

description: Set number of significant digits and format used by printm() and int_printm(). The format can either be PLAIN or MATLAB. MATLAB gives a text output that is easily imported into matlab. PLAIN on the other hand is a bit easier read.

arguments: `int sdig` Number of significant digits.

`int format` Output format style, either MATLAB or PLAIN.

returns: nothing

usage: `init_print(6, MATLAB); // 6 significant digits and MATLAB output format`

see also: `printm(), int_printm(), init_GM()`

file: `utils.c`

**int init_temp_list(void)**

description: Initialize the list for allocation of temporary matrices

arguments: none

returns: `int` NOERR if all right, −1 if out of memory usage: `err = init_temp_list(); /* Initialize temp matrices, inspect err for errors. */`

see also: `init_GM()`

note: init_temp_list() is normally called from init_GM();

file: `matrix.c`

## int_matrix *int_add(int_matrix *a, int_matrix *b)

description: Adds matrices a and b.

arguments: `int_matrix *a, *b` Input matrices.

returns: `int_matrix *` The sum of a and b, NULL if error.

usage:
```
sum = int_add(a, b); /* sum = a + b
matlab equivalent:
>> sum = a + b; */
```

file: `int_matrix.c`

## int_matrix *int_appendcols(int_matrix *a, int_matrix *b)

description: Returns a matrix with b's columns appended to a ([a, b]). Naturally, a and b must have the same number of rows.

arguments: `int_matrix *a, *b` Input matrices.

returns: `int_matrix *` [a, b], NULL if error.

usage:
```
c = int_appendcols(a, b); /* c = [a, b]

matlab equivalent:

>> c = [a, b]; */
```

file: `int_matrix.c`

### int_matrix *int_appendrows(int_matrix *a, int_matrix *b)

description: Returns a matrix with b's rows appended to a ([a; b]). Naturally, a and b must have the same number of columns.

arguments: `int_matrix *a, *b` Input matrices.

returns: `int_matrix *` [a; b], NULL if error.

usage:
```
c = int_appendrows(a, b); /* c = [a; b]

matlab equivalent:

>> c = [a; b]; */
```

file: `int_matrix.c`

### int_matrix *int_assign(int_matrix *target, int_matrix *rows, int_matrix *cols, matrix *source)

description: Puts the matrix source into a sub matrix of target indicated by rows and cols. That is, rows and cols defines a sub matrix of target (exactly like int_sub_matrix()) and this sub matrix is overwritten with data from the source matrix. This is analogous to the matlab statement target(rows, cols) = source. Needless to say, the sub matrix of target and source must be of the same dimensions. For example, suppose

target = [1 2 3 4; 5 6 7 8; 9 10 11 12],

source = [13 14; 15 16],

rows = [3 2] and cols = [1 2], the resulting matrix would be

[ 1 2 3 4; 15 16 7 8; 13 14 11 12 ].

If rows or cols is NULL, this means all the rows and all the columns of target. That is, target(rows,:) = source would be coded as assign(target, rows, NULL, source), and similarly, target(rows,:) = source would be coded as int_assign(target, rows, NULL, source).

| | | |
|---|---|---|
| arguments: | `int_matrix *target` | Matrix to be written to. |
| | `int_matrix *rows` | Row indexing matrix. |
| | `int_matrix *cols` | Column indexing matrix |
| | `int_matrix *source` | Matrix whose data will be written to target. |

returns: `int_matrix *` Copy of target with parts overwritten by source, NULL if error.

usage:
```
int_assign(target, rows, cols, source); // target(rows, cols) =
source
int_assign(target, rows, NULL, source); // target(rows, :)  =
source
```

`int_assign(target, NULL, cols, source); // target(:, cols) = source`

see also: `int_sub_matrix(), int_temp_copy()`

note: Does not handle the case target(:,:) = source. For this case copy source with target = int_temp_copy(source).

file: `int_matrix.c`

int int_cmplx_promote(int_matrix *a, int_matrix *b)

description: Promotes, if necessary, the operands a and b to complex matrices, that is, if either a or b is complex then both a and b are converted to complex matrices.

arguments: `int_matrix *a, *b` Matrices to be promoted.

returns: `int` NOERR if successful, −1 if malloc failed

usage: `res = int_cmplx_promote(a, b); /* Complex promotes a and b, OK if res == NOERR. */`

see also: `int_op_check()`

file: `int_matrix.c`

int_matrix *int_conj(int_matrix *mat)

description: Returns a matrix equal to the complex conjugate of the input matrix.

arguments: `int_matrix *mat` Input matrix.

returns: `int_matrix *` The complex conjugated input matrix

usage: `conj_A = int_conj(A); /* conj_A = -A`

`matlab equivalent:`

`>> conj_A = conj(A); */`

file: `int_matrix.c`

## int_matrix *int_copy_matrix(int_matrix *source)

description: Returns a copy of the matrix source. The copy is allocated with int_new_matrix(). Note that it is the user's responsibility to free this matrix. Use int_copy_temp() to get a temporary matrix (which will be freed by kill_temp_list() or close_GM()).

arguments: `int_matrix *source` Matrix to be copied.

returns: `int_matrix *` Copy of source. NULL if out of memory.

usage: `new = int_copy_matrix(old); // copy the matrix old to the matrix new`

see also: `int_kill_matrix(), int_new_temp(), int_copy_temp(),`

`int_new_matrix(),`

`kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with int_copy_matrix (with int_kill_matrix).

file: `int_matrix.c`

**int_matrix *int_copy_temp(int_matrix *source)**

description: Returns a copy of the matrix source. The copy is allocated with int_new_temp() and therefore, is a temporary matrix. To free ALL temporary matrices, use kill_temp_list() or close_GM().

arguments: `int_matrix *source` Matrix to be copied.

returns: `int_matrix *` Copy of source. NULL if out of memory.

usage: `new = int_copy_temp(old); // copy the matrix old to the temporary matrix new`

see also: `int_kill_matrix()`, `int_copy_matrix()`, `int_new_temp()`, `int_new_matrix()`, `kill_temp_list()`

file: `int_matrix.c`

**int_matrix *int_herm(int_matrix *mat)**

description: Returns the conjugate transpose of the input matrix, that is, takes the hermitian of mat.

arguments: `int_matrix *mat` Input matrix.

returns: `int_matrix *` The conjugate transposed input matrix, NULL if error.

usage: `tran_A = int_herm(A); /* tran_A = A'`

```
matlab equivalent:

>> A = A'; % Note ' not .', that is, conjugate transpose */
```

see also: `int_transp()`

file: `int_matrix.c`

### int_matrix *int_imag(int_matrix *mat)

description: Returns a matrix containing the imaginary part of mat.

arguments: `int_matrix *mat` Input matrix.

returns: `int_matrix *` Imaginary part of mat.

usage: `im_part = int_imag(cmplx_matrix); /* im_part = Im[cmplx_matrix]`

```
matlab equivalent:

>> im_part = imag(cmplx_matrix); */
```

file: `int_matrix.c`

### int_matrix *int_index_cols(int_matrix *mat, int_matrix *ind)

description: Returns the columns of mat that are pointed out by ind. The elements of ind are used to pick out columns. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3 1], then the result would be a matrix of the form [3 2; 6 5; 9 8]. This is analogous to the matlab statement, mat(:, ind).

| | | |
|---|---|---|
| arguments: | `int_matrix *mat` | Matrix to be indexed. |
| | `int_matrix *ind` | Column indexing matrix. |

returns: `int_matrix *` The indexed input matrix, NULL if error.

usage:
```
B = int_index_cols(mat, ind); /* B = mat(:, ind)

matlab equivalent:

>> B = mat(:, ind); */
```

see also: `int_index_rows()`, `int_index_rows_cols()`, `int_sub_matrix()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

### int_matrix *int_index_rows(int_matrix *mat, int_matrix *ind)

description: Returns the rows of mat that are pointed out by ind. The elements of ind are used to pick out rows. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind = [3 1], then the result would be a matrix of the form [7 8 9; 4 5 6]. This is analogous to the matlab statement, mat(ind, :).

| | | |
|---|---|---|
| arguments: | `int_matrix *mat` | Matrix to be indexed. |
| | `int_matrix *ind` | Row indexing matrix. |

returns: `int_matrix *` The indexed input matrix, NULL if error.

usage: `B = int_index_rows(mat, ind); /* B = mat(ind, :)`

```
matlab equivalent:

>> B = mat(ind, :)  */
```

see also: `int_index_cols(), int_index_rows_cols(), int_sub_matrix()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

**int_index_rows_cols(int_matrix *mat, int_matrix *ind_a, int_matrix* ind_b)**

description: Returns the rows and columns of mat that are pointed out by ind_a and ind_b. The elements of ind_a and ind_b are used to pick out rows and columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and ind_a = [3 1] and ind_b = [2], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(ind_a, ind_b).

| | | |
|---|---|---|
| arguments: | `int_matrix *mat` | Matrix to be indexed. |
| | `int_matrix *ind_a` | Row indexing matrix. |
| | `int_matrix *ind_b` | Column indexing matrix. |

returns: `int_matrix *` The indexed input matrix, NULL if error.

usage: `B = int_index_rows_cols(mat, ind_a, ind_b); /* B = mat(ind_a, ind_b)`

```
matlab equivalent:

>> B = mat(ind_a, ind_b); */
```

see also: `int_index_rows(), int_index_cols(), int_sub_matrix()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

## void int_kill_matrix(int_matrix *mat)

description: Frees memory used by mat. Note that mat should have been allocated with int_new_matrix() or int_copy_matrix(). If mat is a temporary matrix, that is, allocated explicitly or implicitly with int_new_temp() or int_copy_temp(), then kill_temp_list() should be used.

arguments: `int_matrix *mat`    Matrix to be freed, must have been allocated by int_new_matrix() or int_copy_matrix()

returns: nothing

usage: `int_kill_matrix(A); // Free memory allocated for A`

see also: `int_copy_matrix(), int_new_temp(), int_copy_temp(),`

`int_new_matrix(),`

`kill_temp_list()`

note: The matrix mat must have been allocated by int_new_matrix() or int_copy_matrix()

file: `int_matrix.c`

**int int_max_index(int_matrix *mat)**

description: Return index to maximal element of mat.

arguments: `int_matrix *mat` Input matrix.

returns: `int` Index to the maximal element of mat.

usage:
```
max_i = int_max_index(mat); /* max(mat) = mat[max_i]

matlab equivalent:

>> max_i = find(mat == max(mat(:))); */
```

file: `int_matrix.c`

**int int_min_index(int_matrix *mat)**

description: Return index to minimal element of mat.

arguments: `int_matrix *mat` Input matrix.

returns: `int` Index to the minimal element of mat.

usage:
```
min_i = int_min_index(mat); /* min(mat) = mat[min_i]

matlab equivalent:

>> min_i = find(mat == min(mat(:))); */
```

file: `int_matrix.c`

## int_matrix *int_minus(int_matrix *mat)

description: Returns a matrix equal to the negative of input matrix.

arguments: `int_matrix *mat` Input matrix.

returns: `int_matrix *` The negated input matrix.

usage:
```
minus_A = int_minus(A); /* minus_A = -A

matlab equivalent:

>> minus_A = -A; */
```

file: `int_matrix.c`

## int_matrix *int_mul(int_matrix *a, int_matrix *b)

description: Multiplication of matrices a and b.

arguments: `int_matrix *a, *b` Input matrices.

returns: `int_matrix *` The product of a and b, NULL if error.

usage:
```
prod = int_pmul(a, b); /* pprod = a * b

matlab equivalent:

>> prod = a * b; */
```

file: `int_matrix.c`

**int int_mul_check(int_matrix *a, int_matrix *b)**

description: Check field type and dimensions. First a and b are complex promoted by int_cmplx_promote(). Returns COMP if a and b have the same dimensions. Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR ed to the returned value. Thus, if type = int_op_check(a,b), then (type & CMPLX) will be true if either a or b is complex, (type & REAL) will be true if both a and b are real, (type & COMP) will be true if dimensions match or if a or b is a scalar, (type & SCAL) will be true if a or b is a scalar.

arguments: `int_matrix *a, *b` matrices to be checked

returns: `int` The bits are set according to the description above.

usage:
```
type = int_mul_check(a, b); /* checks dimensions of a, b.  Bits
in type will be set in type according to the description above */
```

see also: `int_cmplx_promote(), int_op_check()`

note: int_mul_check() is used by int_mul(). A scalar is compatible with any matrix.

file: `int_matrix.c`

**int_matrix *int_new_matrix(int rows, int cols, int type)**

description: Allocates memory for a matrix with dimension rows and cols and of type type (CMPLX or REAL). Note that it is the user's responsibility to free this matrix. Use int_new_temp() to allocate memory for a temporary matrix (which will be freed by kill_temp_list() or close_GM()).

arguments: `int rows, cols` Dimensions of matrix to be allocated

`int type` Type of matrix, CMPLX for a complex matrix and REAL for a real matrix

returns: `int_matrix *` Matrix of the requested size and type. NULL if out of memory.

usage: `mat = int_new_matrix(3, 5, CMPLX); /* Allocate memory for a 3x5 complex matrix */`

see also: `int_copy_matrix()`, `int_new_temp()`, `int_copy_temp()`, `int_kill_matrix()`, `kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with int_new_matrix() (with int_kill_matrix()).

file: `int_matrix.c`

## int_matrix *int_new_temp(int rows, int cols, int type)

description: Allocates memory for a temporary matrix with dimension rows and cols and of type. Note that it is the user's responsibility to free this matrix. To free ALL temporary matrices, use kill_temp_list() or close_GM().

arguments: `int rows, cols` Dimensions of matrix to be allocated.

`int type` Type of matrix, CMPLX for a complex matrix and REAL for a real matrix.

returns: `int_matrix *` Matrix of the requested size and type. NULL if out of memory.

usage: `mat = int_new_temp(3, 5, CMPLX); // Allocate memory for a temporary 3x5 complex matrix`

see also: `int_kill_matrix()`, `int_copy_matrix()`, `int_copy_temp()`, `int_new_matrix()`,

`kill_temp_list()`

file: `int_matrix.c`

**int int_op_check(int_matrix *a, int_matrix *b)**

description: Check field type and dimensions. First a and b are complex promoted by int_cmplx_promote(). Returns COMP if a and b have the same dimensions. Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR ed to the returned value. Thus, if type = int_op_check(a,b), then (type & CMPLX) will be true if either a or b is complex, (type & REAL) will be true if both a and b are real, (type & COMP) will be true if dimensions match, (type & SCAL) will be true if either a or b is a scalar.

arguments: `int_matrix *a, *b` Matrices to be checked

returns: `int` The bits are set according to the description above.

usage: `type = int_op_check(a, b); /* checks dimensions of a, b. Bits in type will be set in type according to the description above */`

see also: `int_cmplx_promote(), int_mul_check()`

note: int_op_check() is used by int_add(), int_pmul(), int_pdiv(). A scalar is compatible with any matrix.

file: `int_matrix.c`

## int_matrix *int_pmul(int_matrix *a, int_matrix *b)

description: Point wise multiplication of matrices a and b.

arguments: `int_matrix *a, *b` Input matrices.

returns: `int_matrix *` The point wise product of a and b, NULL if error.

usage:
```
pprod = int_pmul(a, b); /* pprod = a .* b

matlab equivalent:

>> pprod = a .* b; */
```

file: `int_matrix.c`

## void int_printm(int_matrix *mat)

description: Prints an integer matrix to stdout using the number of significant digits and output style set by init_print().

arguments: `int_matrix *mat` Matrix to be printed

returns: `void`

usage: `printm(Rxx); // prints Rxx to stdout`

see also: `init_print() printm()`

file: `utils.c`

### int_matrix *int_range(INTTYPE from, INTTYPE step, INTTYPE to)

description: Creates a vector like matlab's from:step:to. If step is 0, then the step size is set to 1. For example, int_range(1, 2, 7) results in [1 3 5 7], int_range(3, 0, 5) results in [3 4 5].

arguments: `INTTYPE from` Start value.

`INTTYPE step` Step size.

`INTTYPE to` Stop value.

returns: `int_matrix *` Vector with elements starting at from and stopping at to, spaced by step. NULL if error.

usage: `int_range(from, step, to); // from:step:to`

`int_range(from, 0, to); // from:to`

file: `int_matrix.c`

### int_matrix *int_real(int_matrix *mat)

description: Returns a matrix containing the real part of mat.

arguments: `int_matrix *mat` Input matrix.

returns: `int_matrix *`    Real part of mat.

usage: `real_part = int_real(cmplx_matrix); /* real_part =`

`Re[cmplx_matrix]`

`matlab equivalent:`

`>> real_part = real(cmplx_matrix); */`

file: `int_matrix.c`

**int_matrix *int_scl2mat(INTTYPE re, INTTYPE im, int type);**

description: Creates a 1x1 matrix from the scalar (re + j*im), if type is CMPLX. If type is REAL the imaginary part is ignored.

arguments: `INTTYPE re`    Real part.

`INTTYPE im`    Imaginary part

returns: `int_matrix *`    1x1 matrix with the element (re + j*im). NULL if error.

usage: `scalar_mat = int_scal2mat(3, 2, CMPLX); // scalar_mat(1,1) = 3 +`

`j*2`

`scalar_mat = int_scal2mat(3, 2, REAL); // scalar_mat(1,1) = 3`

**int_sub_matrix(int_matrix *mat, int_matrix *rows, int_matrix* int_matrix *cols);**

description: Returns the rows and columns of mat that are pointed out by (the matrices) rows and cols. The elements of rows and cols are used to pick out rows and columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and rows =

[3 1] and cols = [2], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(rows, cols). To just index rows, like matlab's mat(rows, :), set cols to NULL. Similarly, to just index columns, like matlab's mat(:, cols), set rows to NULL. By using this scheme all indexing can be done with int_sub_matrix().

| | | |
|---|---|---|
| arguments: | `int_matrix *mat` | Matrix to be indexed. |
| | `int_matrix *rows` | Row indexing matrix. |
| | `int_matrix *cols` | Column indexing matrix. |

returns: `int_matrix *` The indexed input matrix, NULL if error.

usage:
```
B = int_sub_matrix(mat, rows, cols); // B = mat(rows, cols)

B = int_sub_matrix(mat, rows, NULL); // B = mat(rows, :)

B = int_sub_matrix(mat, NULL, cols); /* B = mat(:, cols)

matlab equivalent:

>> B = mat(rows, cols); */
```

see also: `int_index_rows(), int_index_cols(), int_index_rows_cols()`

note: For greatest convenience, use int_sub_matrix() for all indexing purposes.

file: `int_matrix.c`

int_matrix *int_transp(int_matrix *mat)

description: Returns the transpose of the input matrix. Note: does not conjugate elements. Use int_herm() for conjugate transpose (hermitian).

arguments: `int_matrix *mat` Input matrix.

returns: `int_matrix *` The transposed input matrix, NULL if error.

usage:
```
tran_A = int_transp(A); /* tran_A = A.'

matlab equivalent:

>> A = A.'; % Note .' not ', that is, does not conjugate */
```

see also: `int_herm()`

file: `int_matrix.c`

## void kill_matrix(matrix *mat)

description: Frees memory used by mat. Note that mat should have been allocated with new_matrix() or copy_matrix(). If mat is a temporary matrix, that is, allocated explicitly or implicitly with new_temp() or copy_temp(), then kill_temp_list() should be used.

arguments: `matrix *mat` Matrix to be freed, must have been allocated by new_matrix() or copy_matrix()

returns: nothing

usage:
```
kill_matrix(A); // Free memory allocated for A
```

see also: `copy_matrix(), new_temp(), copy_temp(), new_matrix(), kill_temp_list()`

note: The matrix mat must have been allocated by new_matrix() or copy_matrix()

file: `matrix.c`

### void kill_temp_list(void)

description: Frees memory allocated by ALL temporary matrices. The temporary matrices are allocated, éxplicitly or implicitly, with new_temp() or copy_temp.

arguments: none

returns: nothing

usage: `kill_temp_list(); // Free memory allocated by all temporary matrices`

see also: `kill_matrix(), copy_matrix(), new_temp(), new_matrix(), copy_temp(), close_GM()`

file: `matrix.c`

### matrix *LV2GM(TD1Hdl re, TD1Hdl im)

description: Copies Labview matrices to Gauss Machine (GM) matrices. No explicit memory allocation is necessary. The memory management is handled internally.

arguments: `TD1Hdl re, im` Handles to Labview matrix data structures.

returns: `matrix *` Gauss machine matrix with data from the input arguments re and im.

usage:
```
A = LV2GM(A_real, A_imag); /* Create a GM matrix with real part
data from A_real and imaginary data from A_imag.  */
```

see also: `GM2LV()`

file: `utils.c`

## int max_index(matrix *mat)

description: Return index to maximal element of mat.

arguments: `matrix *mat` Input matrix.

returns: `int` Index to the maximal element of mat.

usage:
```
max_i = max_index(mat); /* max(mat) = mat[max_i]
matlab equivalent:
>> max_i = find(mat == max(mat(:))); */
```

file: `matrix.c`

## matrix *mfixed2float(int_matrix *mat)

description: Converts a fixed-point matrix to floating-point using the word length and number of fractional bits set by init_conv().

arguments: `int_matrix *mat` Fixed-point matrix to be converted to floating-point.

returns: `matrix *` Floating-point matrix corresponding to the fixed-point matrix.

usage:
```
int_matrix *int_Rxx;

matrix *Rxx;

Rxx = mfixed2float(int_Rxx); // convert int_Rxx to floating-point
```

see also: `init_conv(), fixed2float(), mfloat2fixed(), fixed2float()`

file: `conv.c`

## int_matrix *mfloat2fixed(matrix *mat)

description: Converts a floating-point matrix to fixed-point using the word length and number of fractional bits set by init_conv().

arguments: `matrix *mat` Floating-point matrix to be converted to fixed-point.

returns: `int_matrix *` Integer matrix containing the fixed-point representation of the input floating-point matrix.

usage:
```
int_matrix *int_Rxx;

matrix *Rxx;

Rxx = mfixed2float(int_Rxx); // convert int_Rxx to floating-point
```

see also: `init_conv(), fixed2float(), mfloat2fixed(), fixed2float()`

file: `conv.c`

## matrix *mfloor(matrix *mat)

description: Returns a matrix with truncated elements of mat, that is, performs the floor function (rounding to closest smallest integer) on mat.

arguments: `matrix *mat` Input matrix.

returns: `matrix *` The "floored" input matrix, NULL if error.

usage:
```
int_A = mfloor(A); /* int_A = floor(A)

matlab equivalent:

>> int_A = floor(A); */
```

note: The output is still a floating point matrix, even though the elements are truncated to integers.

file: `matrix.c`

## int min_index(matrix *mat)

description: Return index to minimal element of mat.

arguments: `matrix *mat` Input matrix.

returns: `int` Index to the minimal element of mat.

usage:
```
min_i = min_index(mat); /* min(mat) = mat[min_i]

matlab equivalent:

>> min_i = find(mat == min(mat(:))); */
```

file: `matrix.c`

**matrix *minus(matrix *mat)**

description: Returns a matrix equal to the negative of input matrix.

arguments: `matrix *mat` Input matrix.

returns: `matrix *` The negated input matrix.

usage:
```
minus_A = minus(A); /* minus_A = -A

matlab equivalent:

>> minus_A = -A; */
```

file: `matrix.c`

**matrix *mul(matrix *a, matrix *b)**

description: Multiplication of matrices a and b.

arguments: `matrix *a, *b` Input matrices.

returns: `matrix *` The product of a and b, NULL if error.

usage:
```
prod = pmul(a, b); /* pprod = a * b

matlab equivalent:

>> prod = a * b; */
```

file: `matrix.c`

**int mul_check(matrix *a, matrix *b)**

description: Check field type and dimensions. First a and b are complex promoted by cmplx_promote(). Returns COMP if a and b have the same dimensions. Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR ed to the returned value. Thus, if type = op_check(a,b), then (type & CMPLX) will be true if either a or b is complex, (type & REAL) will be true if both a and b are real, (type & COMP) will be true if dimensions match or if a or b is a scalar, (type & SCAL) will be true if a or b is a scalar.

arguments: `matrix *a, *b` matrices to be checked

returns: `int` The bits are set according to the description above.

usage:
```
type = mul_check(a, b); /* checks dimensions of a, b.  Bits in
type will be set in type according to the description above */
```

see also: `cmplx_promote(), op_check()`

note: mul_check() is used by mul(). A scalar is compatible with any matrix.

file: `matrix.c`

**matrix *new_matrix(int rows, int cols, int type)**

description: Allocates memory for a matrix with dimension rows and cols and of type (CMPLX or REAL). Note that it is the user's responsibility to free this matrix. Use new_temp() to allocate memory for a temporary matrix (which will be freed by kill_temp_list() or close_GM()).

arguments: `int rows, cols` Dimensions of matrix to be allocated

`int type` Type of matrix, CMPLX for a complex matrix and REAL for a real matrix

returns: `matrix *` Matrix of the requested size and type. NULL if out of memory.

usage: `mat = new_matrix(3, 5, CMPLX); /* Allocate memory for a 3x5 complex matrix */`

see also: `copy_matrix()`, `new_temp()`, `copy_temp()`, `kill_matrix()`, `kill_temp_list()`

note: It is the user's responsibility to free any matrix that has been allocated with new_matrix() (with kill_matrix()).

file: `matrix.c`

## matrix *new_temp(int rows, int cols, int type)

description: Allocates memory for a temporary matrix with dimension rows and cols and of type. Note that it is the user's responsibility to free this matrix. To free ALL temporary matrices, use kill_temp_list() or close_GM().

arguments: `int rows, cols` Dimensions of matrix to be allocated.

`int type` Type of matrix, CMPLX for a complex matrix and REAL for a real matrix.

returns: `matrix *` Matrix of the requested size and type. NULL if out of memory.

usage: `mat = new_temp(3, 5, CMPLX); // Allocate memory for a temporary 3x5 complex matrix`

see also: `kill_matrix()`, `copy_matrix()`, `copy_temp()`, `new_matrix()`, `kill_temp_list()`

file: `matrix.c`

### int op_check(matrix *a, matrix *b)

description: Check field type and dimensions. First a and b are complex promoted by cmplx_promote(). Returns COMP if a and b have the same dimensions. Returns COMP — SCAL if either a or b is a scalar. In addition, if a and b are complex, CMPLX is OR ed to the returned value, otherwise REAL is OR ed to the returned value. Thus, if type = op_check(a,b), then (type & CMPLX) will be true if either a or b is complex, (type & REAL) will be true if both a and b are real, (type & COMP) will be true if dimensions match, (type & SCAL) will be true if either a or b is a scalar.

arguments: `matrix *a, *b` Matrices to be checked

returns: `int` The bits are set according to the description above.

usage: `type = op_check(a, b); /* checks dimensions of a, b. Bits in type will be set in type according to the description above */`

see also: `cmplx_promote()`, `mul_check()`

note: op_check() is used by add(), pmul(), pdiv(). A scalar is compatible with any matrix.

file: `matrix.c`

**matrix *pdiv(matrix *a, matrix *b)**

description: Point wise division of matrices a and b.

arguments: `matrix *a, *b` Input matrices.

returns: `matrix *` The point wise division of a and b, NULL if error.

usage:
```
pprod = pdiv(a, b); /* pprod = a ./ b

matlab equivalent:

>> pprod = a ./ b; */
```

file: `matrix.c`

**matrix *pmul(matrix *a, matrix *b)**

description: Point wise multiplication of matrices a and b.

arguments: `matrix *a, *b` Input matrices.

returns: `matrix *` The point wise product of a and b, NULL if error.

usage:
```
pprod = pmul(a, b); /* pprod = a .* b

matlab equivalent:

>> pprod = a .* b; */
```

file: `matrix.c`

**void print_error(void)**

description: Prints the global error string to stderr.

arguments: none

returns: nothing

usage: `print_error(); // Print error string to stderr.`

see also: `get_error(), clear_error(), error()`

file: `utils.c`

**void printm(matrix *mat)**

description: Prints a matrix to stdout using the number of significant digits and output style set by init_print().

arguments: `matrix *mat`   Matrix to be printed

returns: `void`

usage: `printm(Rxx); // prints Rxx to stdout`

see also: `init_print() int_printm()`

file: `utils.c`

## matrix *range(FLOATTYPE from, FLOATTYPE step, FLOATTYPE to)

description: Creates a vector like matlab's from:step:to. If step is 0, then the step size is set to 1. For example, range(1, 2, 7) results in [1 3 5 7], range(3, 0, 5) results in [3 4 5].

| | | |
|---|---|---|
| arguments: | `FLOATTYPE from` | Start value. |
| | `FLOATTYPE step` | Step size. |
| | `FLOATTYPE to` | Stop value. |

returns: `matrix *` Vector with elements starting at from and stopping at to, spaced by step. NULL if error.

usage:
```
range(from, step, to); // from:step:to

range(from, 0, to); // from:to
```

file: `matrix.c`

## matrix *real(matrix *mat)

description: Returns a matrix containing the real part of mat.

arguments: `matrix *mat` Input matrix.

returns: `matrix *` Real part of mat.

usage:
```
real_part = real(cmplx_matrix); /* real_part = Re[cmplx_matrix]

matlab equivalent:

>> real_part = real(cmplx_matrix); */
```

file: `matrix.c`

**matrix *scl2mat(double re, double im, int type);**

description: Creates a 1x1 matrix from the scalar (re + j*im), if type is CMPLX. If type is REAL the imaginary part is ignored.

| | | |
|---|---|---|
| arguments: | `FLOATTYPE re` | Real part. |
| | `FLOATTYPE im` | Imaginary part |
| returns: | `matrix *` | 1x1 matrix with the element (re + j*im). NULL if error. |

usage:
```
scalar_mat = scal2mat(3.14, 2.5, CMPLX); // scalar_mat(1,1) =
3.14 + j*2.5
scalar_mat = scal2mat(3.14, 2.5, REAL); // scalar_mat(1,1) = 3.14
```

**void set_fname(char *fname)**

description: This function is used by the integer matrix algebra routines. When entered these routines calls set_fname() with their own function names as arguments. set_fname() will store the current routine's name and if an overflow/underflow is detected by check(), the function name will be put in front of the error message produced by check(). That is, suppose int_mul() is called, then int_mul() makes the call: set_fname(`int_mul:`). If a overflow occurs inside int_mul(), check() will set the global error message to "int_mul: overflow/underflow". See check() and error() for more details.

arguments: `char *fname` String containing prefix to error messages produced by error()

returns: nothing

usage: `set_fname(Function name); /* error messages will look like: Function name:...  */`

see also: `error(), get_error(), print_error()`

file: `conv.c`

## INTTYPE check(INTTYPE result)

description: Checks to if the result of an operation produces an overflow/underflow. When doing fixed-point arithmetic check() will inspect the current word length to make sure that result is inside the dynamic range. If an overflow or underflow occurs, check() will call error() to set the global error string. Pre pended to the error message is the string set by set_fname(). That is, suppose after the call set_fname(`int_pmul`) an overflow is detected by check(). Then the global error string is set to "in_pmul: overflow/underflow", thereby identifying the routine which produced the overflow/underflow. The result is returned modulo the largest number that can be represented by the current word length. That is, if no overflow or underflow occurs the result is returned unchanged.

arguments: `INTTYPE result` Result of a fixed-point operation that is to be checked for overflow/underflow.

returns: `INTTYPE` The returned value is result mod maxint, where maxint is the largest number representable with the current word length.

usage: `INTTYPE a, b, c; c = check(a+b); // c = (a+b) mod maxint, checks for overflow`

see also: `set_fname(), error(), get_error(), print_error(), init_conv()`

file: `conv.c`

**matrix *sub_matrix(matrix *mat, matrix *rows, matrix *cols);**

description: Returns the rows and columns of mat that are pointed out by (the matrices) rows and cols. The elements of rows and cols are truncated to integers (with mfloor()) and are used to pick out rows and columns, respectively. That is, suppose mat = [1 2 3; 4 5 6; 7 8 9], and rows = [3.14 1.99] and cols = [2.1], then the result would be a matrix of the form [8; 2] (that is, elements (3,2) and (1,2)). This is analogous to the matlab statement, mat(rows, cols). To just index rows, like matlab's mat(rows, :), set cols to NULL. Similarly, to just index columns, like matlab's mat(:, cols), set rows to NULL. By using this scheme all indexing can be done with sub_matrix().

| | | |
|---|---|---|
| arguments: | `matrix *mat` | Matrix to be indexed. |
| | `matrix *rows` | Row indexing matrix. |
| | `matrix *cols` | Column indexing matrix. |
| returns: | `matrix *` | The indexed input matrix, NULL if error. |

usage:
```
B = sub_matrix(mat, rows, cols); // B = mat(rows, cols)

B = sub_matrix(mat, rows, NULL); // B = mat(rows, :)
```

```
B = sub_matrix(mat, NULL, cols); /* B = mat(:, cols)

matlab equivalent:

>> B = mat(rows, cols); */
```

see also: `index_rows(), index_cols(), index_rows_cols()`

note: For greatest convenience, use sub_matrix() for all indexing purposes.

file: `matrix.c`

## matrix *transp(matrix *mat)

description: Returns the transpose of the input matrix. Note: does not conjugate elements. Use herm() for conjugate transpose (hermitian).

arguments: `matrix *mat` Input matrix.

returns: `matrix *` The transposed input matrix, NULL if error.

usage:
```
tran_A = transp(A); /* tran_A = A.'

matlab equivalent:

>> A = A.'; % Note .' not ', that is, does not conjugate */
```

see also: `herm()`

file: `matrix.c`

| Pin Number | Signal Name | Description |
|---|---|---|
| 208—210 | BA0-2 | Board address. |
| 211—213 | PA0-2 | Port address. |
| 299 | XIW* | X-bus input FIFO write signal. |
| 300 | YIW* | Y-bus FIFO write signal. |
| 301 | XOR* | X-bus output FIFO read signal. |
| 302 | AROE* | Accumulator to X-bus transport enable. |
| 303 | XBOE* | X-bus transport output enable. |
| 304 | VECTORMODE | Vector mode signal. |
| 305 | ROE* | Adder RAM output enable. |
| 306 | MROE* | Multiplier RAM output enable. |
| 307 | XIFLRT* | X-bus input FIFO restart. |
| 308 | XIR* | X-bus input FIFO read signal. |
| 309 | YIFLRT* | Y-bus FIFO restart. |
| 310 | YIR* | Y-bus FIFO read signal. |
| 311 | XOFLRT* | X-bus output FIFO restart. |
| 312 | XOW* | X-bus output FIFO write signal. |
| 313 | AREN* | Accumulator to X-bus latch enable. |
| 314 | ARWE* | Adder RAM write strobe. |
| 315 | SREN* | Accumulator latch enable. |
| 316 | RESBWE* | Residue bus write enable. |
| 317 | RESBRE* | Residue bus read enable. |
| 318 | CLR* | Processor wide clear signal. |
| 319 | PREN* | Product latch enable. |
| 320 | YBEN* | Y-bus transport latch enable. |
| 321 | XFEN* | X-bus to multiplier RAM latch enable. |
| 322 | XBEN* | X-bus transport latch enable. |
| 323 | MRWE* | Multiplier RAM write enable. |
| 324 | ARITHMODE | Arithmetic mode signal. |

Table 6.1: Gauss Machine Processor Control Signals

# Chapter 7

# ALGORITHMS

As previously stated, the Gauss machine is designed primarily to perform level 3 operations[11]. It also has a vector mode of operation which handles level 2 and level 1 operations efficiently. Some algorithms for the Gauss machine are described below. Although the Gauss machine operates on GEQRNS operands, the algorithms are discussed using the familiar notation of complex numbers.

## 7.1 Matrix Product Based Algorithms

This section describes the implementation of the matrix multiplication operation on the Gauss machine and implementation of algorithms which are based upon matrix multiplication.

### 7.1.1 Matrix Multiplication

If $\mathbf{A} \in \mathbf{C}^{m\times n}$ and $\mathbf{B} \in \mathbf{C}^{n\times r}$ then $\mathbf{AB} = \mathbf{D}$ where $\mathbf{D} \in \mathbf{C}^{m\times r}$, and each $d_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. The core of the matrix multiplication is the multiply-accumulate operation.

Suppose we wish to multiply two $2 \times 2$ matrices. The data are presented in a sloped data front on the X and Y sides of the array and the results are accumulated in place. This configuration is depicted in Figure 7.1. Note that the processors are configured as simple multiply-accumulate units. Each clock cycle, the sloped $\mathbf{A}$ and $\mathbf{B}$ data fronts are advanced one processing element to the right and up, respectively, and the input operands are multiplied and accumulated.

The configuration shown in Figure 7.1 assumes that the array elements are pre-initialized to zero. The leading zeros keep the array initialized to zero and the trailing zeros maintain the results while the computation is completed. This multiplication uses $2 \times 2$ input matrices, however, it can be extended to input matrices where **A** is $2 \times n$ and **B** is $n \times 2$. Matrix multiplication of larger arrays may be achieved by using block multiplication. Systolic arrays which use in-place accumulation of results are well suited to block operations.

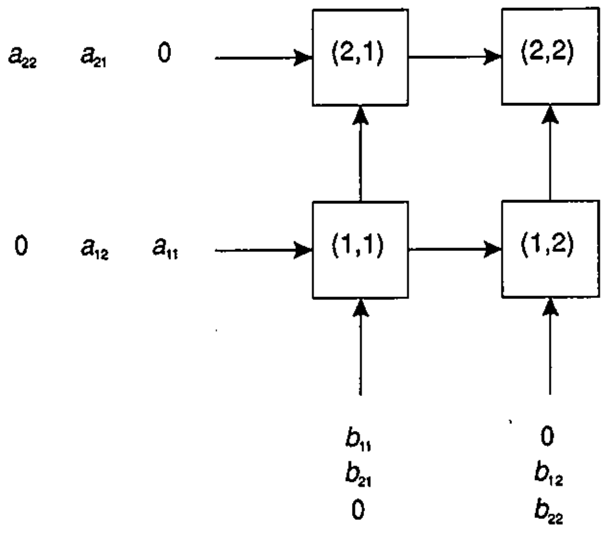


Figure 7.1: Example of Matrix Multiplication

The computation of **AB** where **A** and **B** are of arbitrary dimensions can be performed by decomposing **A** into blocks of two rows and $n$ columns, and by decomposing **B** into blocks of $n$ rows and two columns. This decomposition is depicted below:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \vdots \\ \mathbf{A}_i \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{B}_3 & \cdots & \mathbf{B}_j \end{pmatrix}.$$

where each $\mathbf{A}_k \in \mathbf{C}^{2\times n}$ and each $\mathbf{B}_l \in \mathbf{C}^{n\times 2}$. If a matrix does not have dimensions which meet the multiple of three rows or columns requirements, then the matrix may be padded with zeros. Each of these matrix block products requires $n+4$ cycles to complete the computation and two cycles to remove the results from the array. The result of the matrix block computation is a $2\times 2$ matrix. Thus for $\mathbf{A} \in \mathbf{C}^{k\times n}$ and $\mathbf{B} \in \mathbf{C}^{n\times r}$, the number of cycles required to complete the product $\mathbf{AB}$ is given by

$$O(\mathbf{AB}) = \lceil k/2 \rceil \lceil r/2 \rceil (n+6), \tag{7.1}$$

where $\lceil \bullet \rceil$ represents the greatest integer or ceiling function. Clearly from Equation 7.1, if $n$ is small then the overhead associated with the time required to shift data out and the zero padding becomes significant. As $n \to 1$, the operation degenerates into an outer product. The dynamic range requirements for the matrix multiplication are determined by the size of $\mathbf{A}$ and $\mathbf{B}$, and the dynamic range of the data in the matrices. To be exact, suppose that $\mathbf{A}$ has a dynamic range of $p$, $\mathbf{B}$ has a dynamic range of $q$, and the dimension quantity $n$ is as given above. Then the dynamic range requirements for the product $\mathbf{AB}$ is given by

$$O_D(\mathbf{AB}) = (p+q) + n - 1. \tag{7.2}$$

### 7.1.2 Discrete Fourier Transform

The DFT may be easily expressed as a level 3 operation. The DFT is given by the following:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}.$$

We may express this function in a linear algebraic form by assigning all $x(n)$ to a column vector, $\mathbf{x} \in \mathbf{C}^{N\times 1}$, and the complex exponential to a Vandermonde matrix $\mathbf{W} \in \mathbf{C}^{N\times N}$ where $\mathbf{W}$ is given by

$$\mathbf{W} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ W^{-1} & W^{-2} & W^{-3} & \cdots & W^{-(N-1)} \\ W^{-2} & W^{-4} & W^{-6} & \cdots & W^{-2(N-1)} \\ W^{-3} & W^{-6} & W^{-9} & \cdots & W^{-3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ W^{-(N-1)} & W^{-2(N-1)} & W^{-3(N-1)} & \cdots & W^{-(N-1)^2} \end{pmatrix},$$

where $W = e^{j2\pi/N}$. Thus the DFT of $\mathbf{x}$ is given by

$$\mathbf{X} = \mathbf{W}^T\mathbf{x}$$

Where $\mathbf{X} \in \mathbf{C}^{N\times 1}$. Clearly $\mathbf{W}$ may be precomputed and its form depends upon the number of points to be computed and the dynamic range requirements of the problem. Obviously, since the Gauss machine performs level 3 operations at greatest efficiency when the dimensions of the operand matrices are multiples of two it may be desirable to gang three DFTs together by replacing the column vector $\mathbf{x}$ with an $N \times 2$ matrix where each column of data represents a vector to be transformed. Additionally, if only selected frequency bands are of interest then the DFT need only be computed for those bands.

### 7.1.3 Convolution and Correlation

Like the DFT, convolution and correlation may be expressed as level 3 operations. Recall that discrete correlation and convolution are given by

$$X(\tau) = \sum_n x(n)y(n+\tau),$$
$$Y(\tau) = \sum_n x(n)y(\tau - n),$$

respectively. We will develop the correlation example here; convolution follows directly. Let $x$ and $y$ be data sequences of length $N$ and $M$, respectively. The correlation of $x$ and $y$ will have a length of $N + M - 1$. We will assign the sequence $x$ to a row vector $\mathbf{x} \in \mathbf{C}^{1\times(N+M-1)}$ given by

$$\mathbf{x} = \left( \begin{array}{ccccc|ccc} x_0 & x_1 & x_2 & \cdots & x_{N-1} & 0 & \cdots & 0 \end{array} \right).$$

The sequence $y$ is used to build a matrix $\mathbf{Y} \in \mathbf{C}^{N+M-1\times N+M-1}$, which when multiplied with $\mathbf{x}$ will give the correlation sequence of $x$ and $y$. Since $\mathbf{x}$ is a row vector we would like the columns of $\mathbf{Y}$ to be shifted versions of the sequence $y$. Assuming $y = \{y_0, y_1, y_2, \ldots, y_{M-1}\}$, then the matrix $\mathbf{Y}$ would have the form

$$\mathbf{Y} = \begin{pmatrix} y_{M-1} & y_{M-2} & y_{M-3} & \cdots & 0 & 0 & 0 \\ 0 & y_{M-1} & y_{M-2} & \cdots & 0 & 0 & 0 \\ 0 & 0 & y_{M-1} & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & y_1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & y_2 & y_1 & 0 \\ 0 & 0 & 0 & \cdots & y_3 & y_2 & y_1 \end{pmatrix}.$$

Thus, we can see that the product $\mathbf{x}\mathbf{Y}$ will produce the correlation sequence of $x$ and $y$. The above form of the correlation is suboptimal in that it uses level 2 operations to perform the correlation operation. Optimal performance is obtained by utilizing level

3 operations to perform the correlation. We will form a new matrix $\mathbf{X} \in \mathrm{C}^{2\times N+M-1}$ which contains a shifted version of the row vector $\mathbf{x}$:

$$\mathbf{X} = \begin{pmatrix} x_0 & x_1 & x_2 & \cdots & x_{N-1} & 0 & 0 & \cdots & 0 \\ 0 & x_0 & x_1 & x_2 & \cdots & x_{N-1} & 0 & \cdots & 0 \end{pmatrix}.$$

The form of the matrix $\mathbf{X}$ requires that the form of $\mathbf{Y}$ be modified so that it reflects the shifts in $\mathbf{X}$. The first column of the new version of $\mathbf{Y}$, $\mathbf{Y}'$, will produce the first three elements of the correlation sequence; the second column will produce the next three, and so forth. The form of $\mathbf{Y}'$ is given as

$$\mathbf{Y}' = \begin{pmatrix} y_{M-3} & y_{M-6} & \cdots & 0 & 0 \\ y_{M-2} & y_{M-5} & \cdots & 0 & 0 \\ y_{M-1} & y_{M-4} & \cdots & 0 & 0 \\ 0 & y_{M-3} & \cdots & 0 & 0 \\ 0 & y_{M-2} & \cdots & 0 & 0 \\ 0 & y_{M-1} & \cdots & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \cdots & y_1 & 0 \\ 0 & 0 & \cdots & y_2 & 0 \\ 0 & 0 & \cdots & y_3 & 0 \\ 0 & 0 & \cdots & y_4 & y_1 \\ 0 & 0 & \cdots & y_5 & y_2 \\ 0 & 0 & \cdots & y_6 & y_3 \end{pmatrix}.$$

The product has the form $\mathbf{X}\mathbf{Y}' \in \mathrm{C}^{2\times\lceil(N+M-1)/2\rceil}$. This level 3 form of the correlation operation executes two times faster than the level 2 version, and the computational speed of the level 3 algorithm will grow geometrically with $N$ for an $N \times N$ array, while the level 2 version will only have a linear growth of the computational speed on the same array. Additionally, since $\mathbf{Y}'$ is banded the structure can be exploited to avoid mass multiplication by zeros.

## 7.2 Vector Mode Algorithms

While many algorithms may be expressed as level 3 operations, there are several operations which may not be performed efficiently on the Gauss machine while operating in systolic mode. In order to improve this situation, an architectural enhancement was made to allow the Gauss machine to operate in vector mode. The vector mode of operation uses a subset of the Gauss machine's processing elements and a minimal amount of additional hardware to form the vector processor. See Figure 6.3 and Figure 6.4.

### 7.2.1 Vector Addition

A common operation is the addition of two vectors. While the systolic mode of operation can be used accumulate two vectors together, it is very inefficient. Let $\mathbf{x}, \mathbf{y} \in \mathbf{C}^N$ be two vectors which are to be added. In order to add them in the systolic mode of operation of the array, they are appended so as to form a matrix $\mathbf{Z} \in \mathbf{C}^{N \times 2}$ such that

$$\mathbf{Z} = (\mathbf{x}|\mathbf{y}).$$

The matrix $\mathbf{Z}$ is then multiplied by the matrix $(11)$ to form the sum of the two vectors:

$$\mathbf{x} + \mathbf{y} = (11)\,\mathbf{Z}.$$

This technique may be extended to the accumulation of $K$ vectors, leading to the form

$$\sum_{i=0}^{K-1} \mathbf{x}_i = \underbrace{(111\cdots 1)}_{K}(\mathbf{x}_0|\mathbf{x}_1|\mathbf{x}_2|\cdots \mathbf{x}_{K-1}).$$

From Equation 7.1 we know that the number of cycles required to complete the product is

$$\lceil K/2 \rceil (N+6),$$

which produces a total of $(K-1)N$ complex addition operations. This leads to an efficiency metric for systolic mode vector addition given by

$$\eta_{AS} = \frac{(K-1)N}{\lceil K/2 \rceil (N+6)} \text{complex additions per cycle.}$$

In the vector mode of addition, the summands are accumulated after being multiplied by one. The result is that the number of cycles required for the addition of $K$ vectors of length $N$ is given by

$$KN+3,$$

thus leading to an efficiency metric of

$$\eta_{AV} = \frac{2(K-1)N}{KN+3} \text{complex additions per cycle,}$$

where the factor of 2 is a result of operating two PEs in tandem as vector accumulators. For large $N$, and $K$ even, $\eta_{AS} \approx \eta_{AV}$. However, for $N$ large and $K$ odd, then

$$\frac{\eta_{AV}}{\eta_{AS}} = \frac{K+1}{K}.$$

In general, we would expect the vector method to have slightly better performance than the systolic method.

### 7.2.2 Pointwise Vector Multiplication

Pointwise multiplication of vectors is an important operation in signal processing applications. Pointwise multiplication of vectors is used to window vectors, and to scale a vector. In systolic mode, a vector of length $N$ may be pointwise multiplied by another vector by taking the second vector and placing it on the diagonal of an $N \times N$ matrix of zeros and then multiplying it with the first vector as follows. Suppose $\mathbf{x}, \mathbf{y} \in \mathbf{C}^N$. Then the pointwise vector product (denoted by .*) of these two vectors is

$$\mathbf{x}. * \mathbf{y} = \begin{pmatrix} x_0 & 0 & 0 & \cdots & 0 \\ 0 & x_1 & 0 & \cdots & 0 \\ 0 & 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & x_{N-1} \end{pmatrix} \mathbf{y}.$$

This expression may be evaluated in a manner which reduces the obvious large order of computation by just producing the block matrix products of the vector and the $2 \times 2$ submatrices centered along the diagonal. This produces an expression for the number of cycles required to execute the pointwise vector product given as $\lceil N/2 \rceil 8$ cycles. Thus, since this is the number of cycles required to execute $N$ multiplications, the efficiency metric is given as

$$\eta_{MS} = \frac{N}{\lceil N/2 \rceil 8} \text{complex multiplications per cycle.}$$

In vector mode, the two vector PEs may multiply pairs of multiplicands in a pipelined manner, one product per cycle per PE. Thus, it requires $\lceil N/2 \rceil$ cycles to pointwise multiply two vectors of length $N$. This leads to an efficiency metric of

$$\eta_{MV} = \frac{N}{\lceil N/2 \rceil} \text{complex multiplications per cycle.}$$

For large $N$, the vector mode of operation is approximately eight times faster than the systolic mode of operation.

## 7.3 QR Decomposition

An important application of the RNS is the solution of linear equations of the form

$$\mathbf{Ax} = \mathbf{b}.$$

Methods such as Gaussian elimination, while amenable to implementation on a vector machine, are not as robust as the QR decomposition (QRD). The QR decomposition is given below.

**Theorem 2 (QR Factorization)** *If* $\mathbf{A} \in \mathbf{C}^{m\times n}$ *of rank* $n$*, then* $\mathbf{A}$ *can be factored into a product* $\mathbf{QR}$*, where* $\mathbf{Q} \in \mathbf{C}^{m\times n}$*, and is a matrix with orthonormal columns, and* $\mathbf{R} \in \mathbf{C}^{n\times n}$ *and is upper triangular and invertible.*

The (QRD) can be performed efficiently in the RNS using the Householder reflection. The Householder reflection is preferable over the Givens rotation since it does not contain transcendental functions. Since the RNS is a division-free integer system of arithmetic, it cannot compute transcendental functions efficiently. Additionally, Givens rotations suffer from potential finite-precision error conditions to which the Householder reflections can be made immune. Finally, the Householder reflection is inner-product rich, thus making it ideal for Gauss machine implementation.

### 7.3.1 Householder Reflections

The Householder reflection is an orthogonal vector transformation which is used to selectively introduce zeros in a vector by reflecting the vector through a plane. In general, the Householder reflection is very efficient when used to introduce a large number of zeros into a vector.

The Householder reflection introduces zeros into a vector $\mathbf{x} \neq 0$ via an orthogonal transformation. The transformation matrix, $\mathbf{H}$ is defined by

$$\mathbf{H} = \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}},$$

where $\mathbf{v}$, the Householder vector is defined by

$$\mathbf{v} = \mathbf{x} \pm \|\mathbf{x}\|_2 e_1. \tag{7.3}$$

When the Householder matrix is applied to the vector $\mathbf{x}$, we arrive at

$$\mathbf{H}\mathbf{x} = \left(\mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}\right)\mathbf{x} = \mp \|\mathbf{x}\|_2 e_1.$$

In computing the Householder vector, $\mathbf{v}$, we may choose to use either of the forms

$$\mathbf{v} = \mathbf{x} + \|\mathbf{x}\|_2 e_1, \text{ or}$$
$$\mathbf{v} = \mathbf{x} - \|\mathbf{x}\|_2 e_1.$$

It is desirable to keep the 2-norm of the Householder vector from becoming small since it would result in the scalar vector $2/\mathbf{v}^T\mathbf{v}$ from Equation 7.3 having a large relative error. Thus we may choose $\mathbf{v}$ in such a way as to maximize the 2-norm of $\mathbf{v}$:

$$\mathbf{v} = \mathbf{x} + \operatorname{sign}(x_1)\|\mathbf{x}\|_2 e_1.$$

Since the RNS is a division-free system of arithmetic, reduction of division operations is attractive. Usually, when the Householder transform is used, it is typically used *en masse*. Thus, it is desirable to maintain the individual Householder transform matrix as a quotient of the form

$$\frac{\mathbf{v}^T\mathbf{v}\mathbf{I} - 2\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}},$$

leading to the form

$$\mathbf{H} \longleftrightarrow \left(\mathbf{v}^T\mathbf{v}, \mathbf{H}'\right),$$

where

$$\begin{aligned} \mathbf{H}' &= \mathbf{v}^T\mathbf{v}\mathbf{H} \\ &= \mathbf{v}^T\mathbf{v}\mathbf{I} - 2\mathbf{v}\mathbf{v}^T. \end{aligned} \tag{7.4}$$

This form avoids division, at the expense of consumption of dynamic range. In particular, if $\mathbf{x}$ is of length $N$, and the dynamic range of the elements of $\mathbf{x}$ is $p$, then the dynamic range of $\mathbf{v}$ is $(N+1)p$. From this we have the dynamic range of $\mathbf{v}^T\mathbf{v}$ given by $(N+1)p + N - 1$. Assuming that $N > 2$, then we may approximate the dynamic range of $\mathbf{H}'$ as $(N+1)p + N - 1$. The next section discusses the application of the Householder transform to the problem of the QRD.

### 7.3.2 Householder QR Factorization

The previous section introduced the use of the Householder transform for the introduction of zeros into a vector. By repeated application of the Householder transformation, we may decompose a matrix $\mathbf{A}$ into an orthogonal matrix and an upper-right triangular matrix, as discussed in Theorem 2 in Section 7.3. The following discussion will examine the implementation of the QRD using the Householder transformation, with special emphasis on the implications of using the transform on the Gauss machine.

Suppose we have $\mathbf{A} \in \mathbf{R}^{m \times n}$, and wish to produce the QR decomposition of $\mathbf{A}$. Then we must generate a series of orthogonal transforms which, when applied to $\mathbf{A}$, will reduce $\mathbf{A}$ to upper-right triangular form. Define a series of orthogonal transforms

$$\tilde{\mathbf{H}}_i = \begin{bmatrix} \mathbf{I}_i & 0 \\ 0 & \mathbf{H}'_i \end{bmatrix},$$

where $\mathbf{I}_i$ is the identity matrix of order $i$, and $\mathbf{H}'_i$ is of the form of Equation 7.4,

$$\mathbf{H}'_i = \mathbf{v}^{(i)T}\mathbf{v}^{(i)}\mathbf{I} - 2\mathbf{v}^{(i)}\mathbf{v}^{(i)T}, \tag{7.5}$$

where $\mathbf{v}^{(i)}$ is defined to be the sub-diagonal entries of the $i$th column of $\mathbf{A}$. Thus, we may successively apply the $\tilde{\mathbf{H}}_i$'s to $\mathbf{A}$:

$$\tilde{\mathbf{H}}_n\tilde{\mathbf{H}}_{n-1}\tilde{\mathbf{H}}_{n-2}\cdots\tilde{\mathbf{H}}_1\mathbf{A},$$

and thus,

$$\mathbf{A} = \mathbf{QR},$$

where

$$\mathbf{Q} = \frac{\tilde{\mathbf{H}}_n\tilde{\mathbf{H}}_{n-1}\tilde{\mathbf{H}}_{n-2}\cdots\tilde{\mathbf{H}}_1}{\prod_{i=1}^{n}\mathbf{v}^{(i)T}\mathbf{v}^{(i)}}.$$

### 7.3.3 Dynamic Range Requirements of the Householder QRD

Since an overflow condition cannot be detected in an RNS system, it is necessary to determine the dynamic range requirements of a given algorithm before it may be used with confidence. Computations must proceed assuming a worst-case set of input data, unless, some occasional error is acceptable.

As in the previous section, suppose we have a matrix $\mathbf{A} \in \mathbf{R}^{m\times n}$, and we truncate $\mathbf{A}$ to some finite precision where we may represent each $a_{ij}$ with $k$ or fewer bits, for all $i \in \{1,2,3,\ldots,m\}$, $j \in \{1,2,3,\ldots,n\}$.

To determine $\mathbf{H}'_i$ we examine Equation 7.5. The first term contains an inner product which acts as a coefficient to an identity matrix. The vector $\mathbf{v}^{(i)}$ is determined using Equation 7.3. The dynamic range of $v_1^{(i)}$ is $k+1$ bits while $v_j^{(i)}$ has a dynamic range of $k$ bits for all $j \in \{2,3,4,\ldots,(m-i)\}$. Thus the dynamic range of the inner product of Equation 7.5 is $2(k+1)+\lceil (m-i-1)/3 \rceil$.

*Proof:*

Let $\mathbf{v} \in \mathbf{R}^n$. Then

$$\mathbf{v}^T\mathbf{v} = v_1^2 + v_2^2 + v_3^2 + \cdots + v_n^2,$$

where $v_1$ is known to have a dynamic range of $k+1$ bits, and all $v_i$ have a dynamic range of $k$ bits, for $i \in \{1,2,3,\ldots,n\}$. Thus, $O\left(v_1^2\right) = 2(k+1)$ and $O\left(v_i^2\right) = 2k$. Examining the above summation, we see that the dynamic range is easily computed:

$$\underbrace{\overbrace{v_1^2}^{2k+2} + \underbrace{\underbrace{\overbrace{v_2^2}^{2k} + \overbrace{v_3^2}^{2k}}_{2k+1} + \overbrace{v_4^2}^{2k}}_{2k+2}}_{(2k+2)+1} + \cdots.$$

This leads to a dynamic range bound of $2(k+1)+\lceil (n-1)/3 \rceil$.

The second term of Equation 7.5 is a scaled outer product. Let $\mathbf{B} = 2\mathbf{v}^{(i)}\mathbf{v}^{(i)T}$. Then $b_{11}$ has dynamic range $2(k+1)+1 = 2k+3$ while $b_{1p}$ and $b_{j1}$ have dynamic range $(k+1)+k+1 = 2(k+1)$, and $b_{jp}$ has dynamic range $k+k+1 = 2k+1$, for all $j \neq 1$, and all $p \neq 1$. To summarize these findings, the dynamic range of $\mathbf{B}$ is given as

$$O_D(\mathbf{B}) = \left( \begin{array}{c|ccc} 2k+3 & 2(k+1) & \cdots & 2(k+1) \\ \hline 2(k+1) & 2k+1 & \cdots & 2k+1 \\ \vdots & \vdots & \ddots & \vdots \\ 2(k+1) & 2k+1 & \cdots & 2k+1 \end{array} \right). \tag{7.6}$$

Finally, the two terms of Equation 7.5 are subtracted leading to the final result for the dynamic range of $\mathbf{H}_i'$. Clearly, since the first term is a scaled identity matrix, the dynamic range of the off-diagonal elements will be as given in Equation 7.6. For

$h'_{i_{11}}$, if $(m-i) \leq 3$ then the dynamic range is $2k+4$, else, if $(m-i) > 3$ then the dynamic range is $2(k+1) + \lceil (m-i)/3 \rceil$. For the remaining diagonal elements, $h'_{i_{jj}}$, $j \neq 1$, the dynamic range is $2(k+1) + \lceil (m-i)/3 \rceil$. To summarize,

$$O_D(\mathbf{H}'_i) = \left( \begin{array}{c|ccccc} a_i & 2(k+1) & \cdots & \cdots & 2(k+1) \\ \hline 2(k+1) & b_i & 2k+1 & \cdots & 2k+1 \\ \vdots & 2k+1 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 2k+1 \\ 2(k+1) & 2k+1 & \cdots & 2k+1 & b_i \end{array} \right),$$

where

$$a_i = \begin{cases} 2k+4 & (m-i) \leq 3 \\ 2(k+1) + \lceil (m-i)/3 \rceil & (m-i) > 3 \end{cases},$$

and $b_i = 2(k+1) + \lceil (m-i)/3 \rceil$.

Clearly, the dynamic range of the above approach quickly gets out of hand. An alternative approach is suggested by [11]. This approach relies on a block representation and is given below.

```
$\mathbf{Y} = \mathbf{v}^{(1)}$
$\mathbf{W} = -2\mathbf{v}^{(1)}/\mathbf{v}^{(1)T}\mathbf{v}^{(1)}$
for $j$=2:$r$
    $\mathbf{z} = -2(\mathbf{I} + \mathbf{W}\mathbf{Y}^T)\mathbf{v}^{(j)}/\mathbf{v}^{(j)T}\mathbf{v}^{(j)}$
    $\mathbf{W} = [\mathbf{W}\mathbf{z}]$
    $\mathbf{Y} = [\mathbf{Y}\mathbf{v}^{(j)}]$
end
```

Let the Householder vectors $\mathbf{v}^{(i)}$ be used to pre-generate $\hat{\mathbf{v}}^{(i)} = -2\mathbf{v}^{(i)}/\mathbf{v}^{(i)T}\mathbf{v}^{(i)}$. Then the above algorithm is modified to take the form given below.

```
$\mathbf{Y} = \mathbf{v}^{(1)}$
$\mathbf{W} = \hat{\mathbf{v}}^{(1)}$
for $j$=2:$r$
    $\mathbf{z} = (\mathbf{I} + \mathbf{W}\mathbf{Y}^T)\hat{\mathbf{v}}^{(j)}$
    $\mathbf{W} = [\mathbf{W}\mathbf{z}]$
    $\mathbf{Y} = [\mathbf{Y}\mathbf{v}^{(j)}]$
end
```

Clearly the modified algorithm is relatively rich in level 3 operations. As the index $j$ increases, the order of computation for the outer product $\mathbf{W}\mathbf{Y}^T$ increases as the number of columns of $\mathbf{W}$ and $\mathbf{Y}$ increases, and thus the processor utilization increases.

Chapter 8

# SUMMARY AND CONCLUSIONS

## 8.1 Motivation

There is a demonstrable need for high speed front-end signal processors for signal and image processing applications. There exist a number of problems (*e.g.*, RADAR, communications, video processing) which demand a level of performance which exceeds the capabilities of the current generation of DSP microprocessors by an order of magnitude or more. This need for speed cannot exacerbate existing constraints on size, power, reliability, and cost. With this motivation the construction of a prototype array of processors based upon the GEQRNS was undertaken. The goal of this prototype was to demonstrate that an array of RNS-based processors could be used to obtain high computational throughputs without exacerbating the aforementioned problems. The Gauss machine was constructed using discrete components as a prototype to a VLSI implementation.

Another goal of the Gauss machine was to demonstrate that an array processor could be constructed which would be applicable to a rich set of problems and thus demonstrate that a technology which was sub-optimal in the architectural sense could be used for a variety of problems, thus affecting a savings in non-recurring engineering costs (NREs). Most signal processing problems are rich in inner products which may be expressed in terms of level 1, level 2, or level 3 operations. The Gauss machine was designed to achieve a high level of efficiency in performing level 3 operations, and a lesser, although still excellent, throughput in level 2 and level 1 operations. Many signal processing problems can be efficiently stated in terms of level 3 operations, and thus this bias towards level 3 operations was created.

## 8.2 Results

There were a number of problems which had to be solved in order to construct the Gauss machine. Initially, certain problems in developing an experimental environment were identified. These problems included packaging constraints, flexibility, and portability. Packaging constraints were solved by developing our own prototyping environment, the InvestiGATOR. The InvestiGATOR also solved the problems of portability and flexibility by incorporating a general purpose computer based upon the Motorola 68030 microprocessor, and the inclusion of the SCSI interface for high speed, portable communications, and the RS-232C interface for low speed, portable communications. The InvestiGATOR solved mechanical packaging constraints encountered in earlier efforts by the construction of a large backplane based system. The backplane includes broadcast and near-neighbor communications making it suitable for a variety of prototyping tasks. Additionally, mechanical constraints were reduced to two dimensions (thickness and length) from the three dimensions (thickness, length, and height) found in conventional environments.

The Gauss machine was constructed on six boards which reside on the InvestiGATOR backplane. Each of these six boards has a $2 \times 2$ array of seven-bit GEQRNS processors. These processors were constructed using low-cost, commodity discrete logic components. The arithmetic elements were implemented with low-cost $32K \times 8$ 85 ns SRAM. The arithmetic elements (SRAMs) are the limiting factor in the speed of the system: the 85 ns SRAMs are suitable for a 10 MHz clock rate, while 35 ns

SRAMs would be suitable for a greater than 20 MHz clock rate, and 15 ns SRAMs could be used to generate a clock rate of 50 MHz. At the 10 MHz clock rate, the array achieves an equivalent peak arithmetic rate of 320 million operations per second when performing complex arithmetic, compared with conventional processors. Each of the processor elements on the board occupies approximately 4.4 $\text{in}^2$ of board area and is a discrete implementation of a structure which occupies only approximately 2 $\text{mm}^2$ when implemented in the MOSIS 2.0 $\mu$m scalable CMOS process.

The six processor boards may be configured to act as a single GEQRNS array which can process arithmetic word widths of approximately twenty-one bits (20.25 bits, or 122 dB). Alternately, the Gauss machine may be configured as a single conventional RNS array processor with a dynamic range of approximately thirty-three bits. Additionally, the array processor may be configured to operate as a vector processor using a subset of the processing elements in the array. While the whole array is ideal for level 3 operations, it cannot perform level 2 and level 1 operations efficiently. To solve this problem, a vector sub-processor was carved out of the array. The vector sub-processor can be used to efficiently perform level 2 and level 1 operations such as matrix-vector, and vector-vector inner products, as well as pointwise addition and multiplication. In the vector processor mode, the peak arithmetic rate is equivalent to 160 million operations per second when performing complex arithmetic, compared to conventional processors.

One of the original goals of the project that was dropped due to budgetary constraints was the construction of forward conversion and CRT engine in hardware. The hardware implementation was replaced by a high performance software implementation which runs on the InvestiGATOR communication processor. While this loss does prevent the Gauss machine from being used for high speed real-time applications, it does not seriously interfere with the goals to be demonstrated since the Gauss machine processor array does demonstrate high arithmetic rates.

In conclusion, the Gauss machine demonstrates a high performance, high RNS content architecture for signal processing applications. The Gauss machine performs at an equivalent peak processing rate of 320 million operations per second when performing complex arithmetic, compared to conventional processors. The Gauss machine demonstrates fault tolerance at an architectural level due to the properties of the RNS. This discrete implementation of the Gauss machine demonstrates a cost parity with conventional, off the shelf technologies, however, substantial cost savings can be expected in a VLSI version of this technology, even when produced for relatively short production runs. The Gauss machine also demonstrates an architecture which can potentially be scaled into other technologies (*e.g.*, ECL and GaAs) to produce performance which exceeds that of the Gauss machine by an order of magnitude or more, thus yielding performance several orders of magnitude greater than that possible with conventional signal processing technology.

# Appendix A

## INVESTIGATOR SCHEMATICS

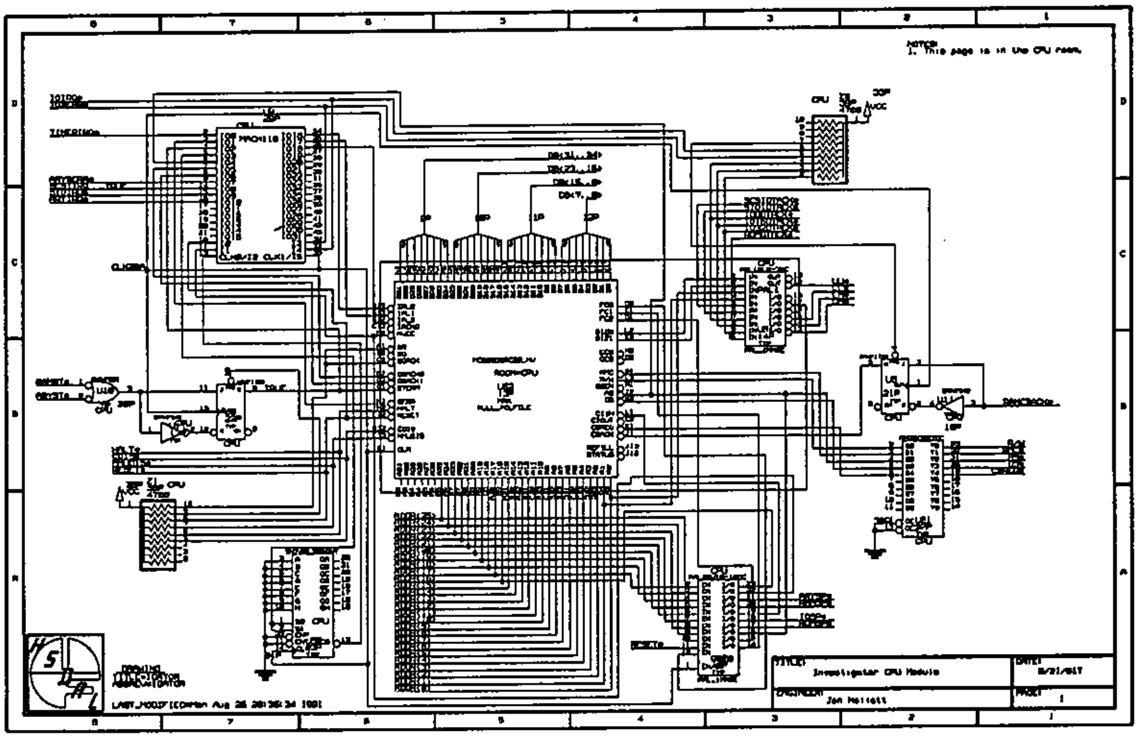


Figure A.1: InvestiGATOR CPU Module

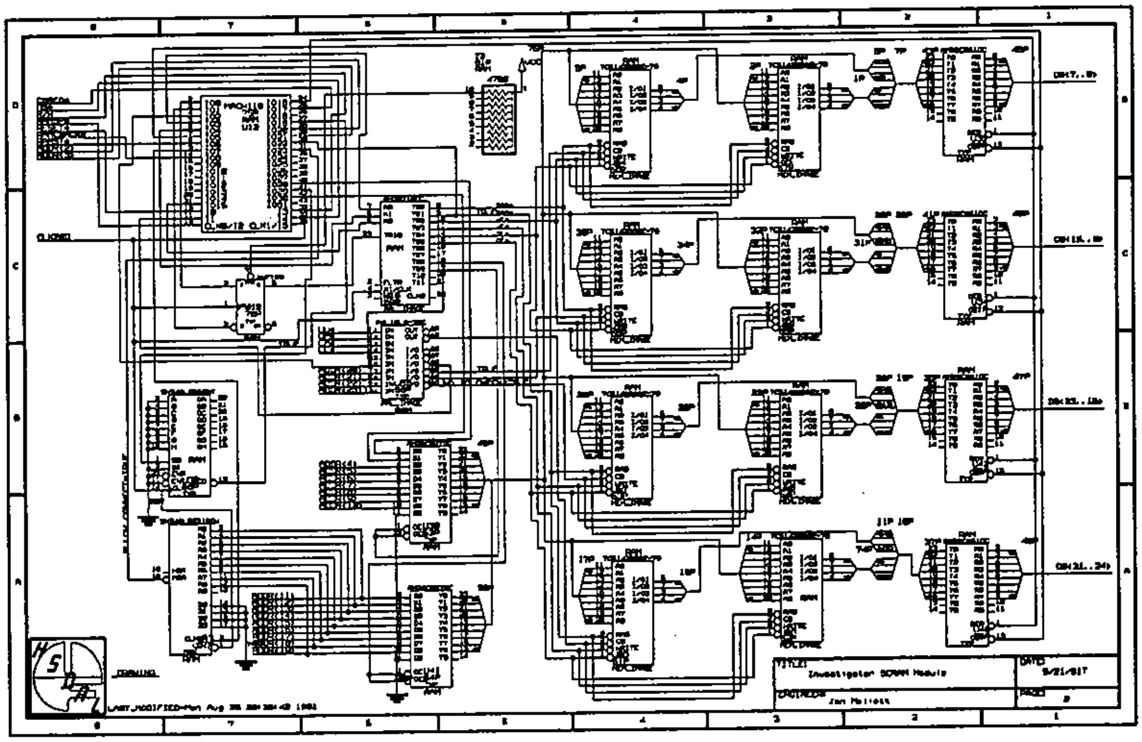


Figure A.2: InvestiGATOR SCRAM Module

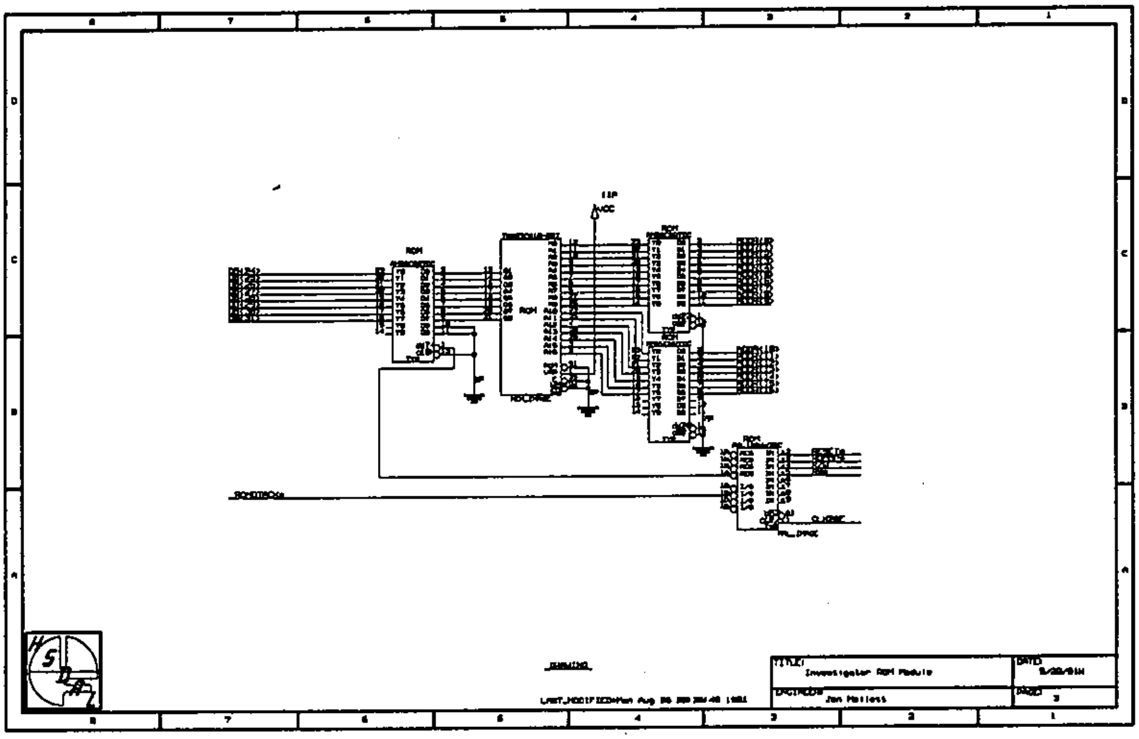


Figure A.3: InvestiGATOR ROM Module

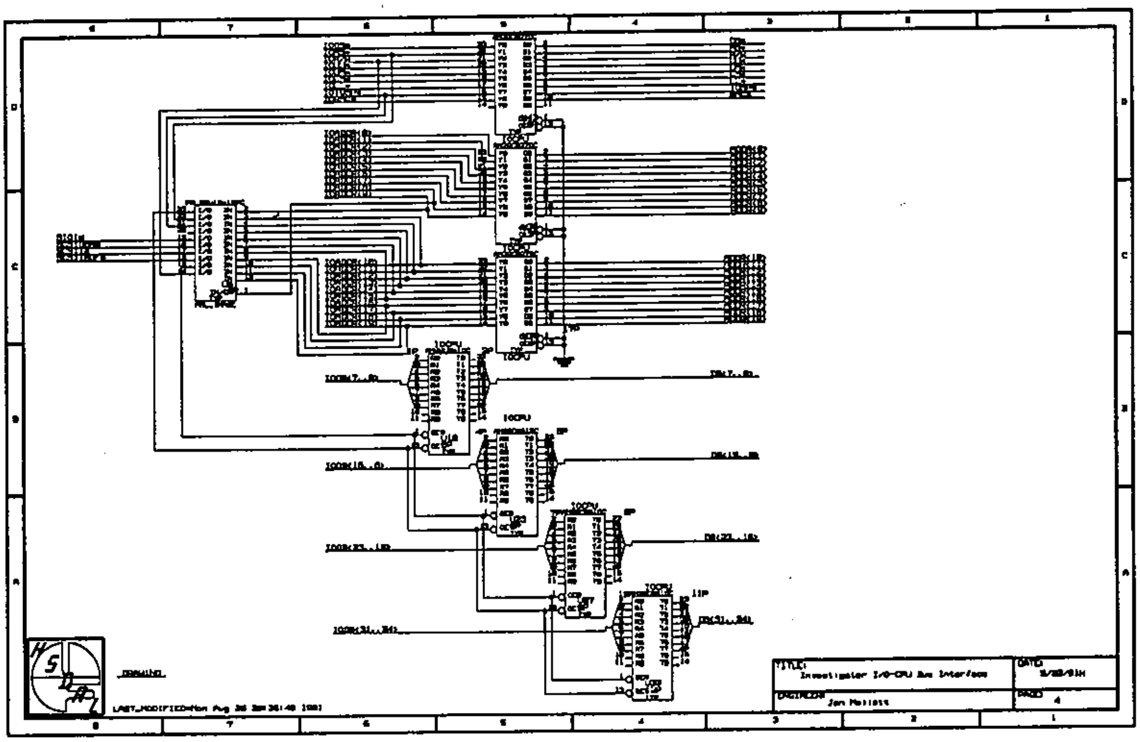


Figure A.4: InvestiGATOR I/O Module

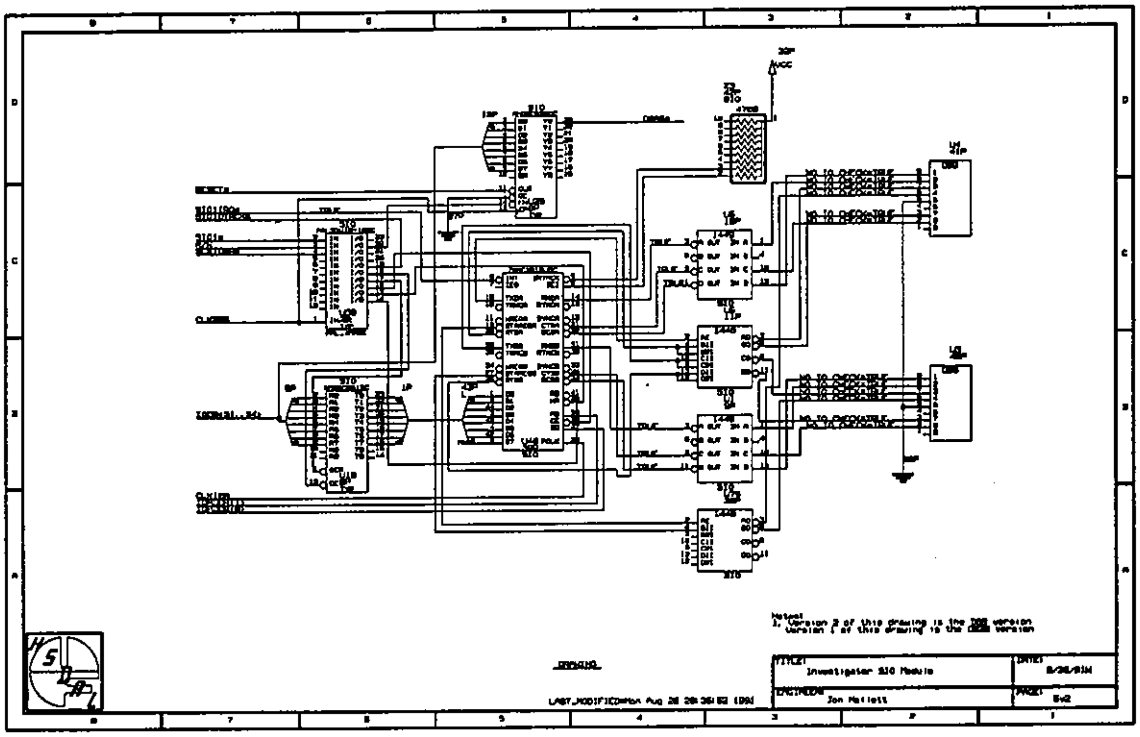


Figure A.5: InvestiGATOR SCSI Module

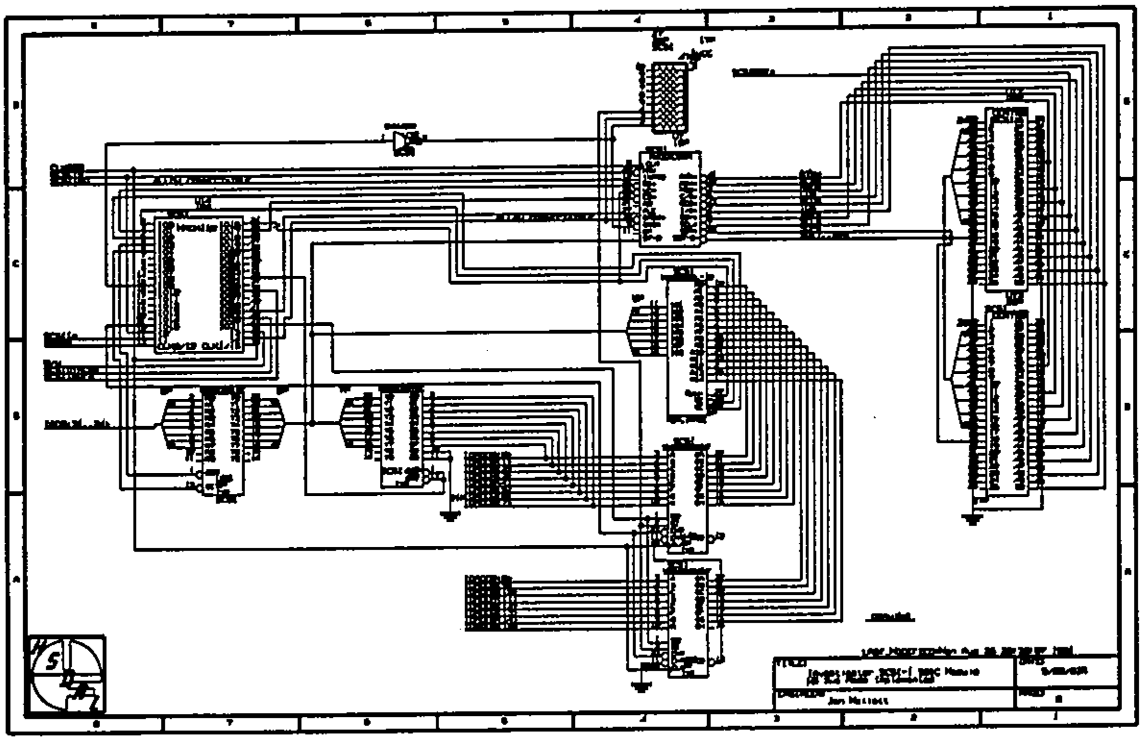


Figure A.6: InvestiGATOR SIO Module

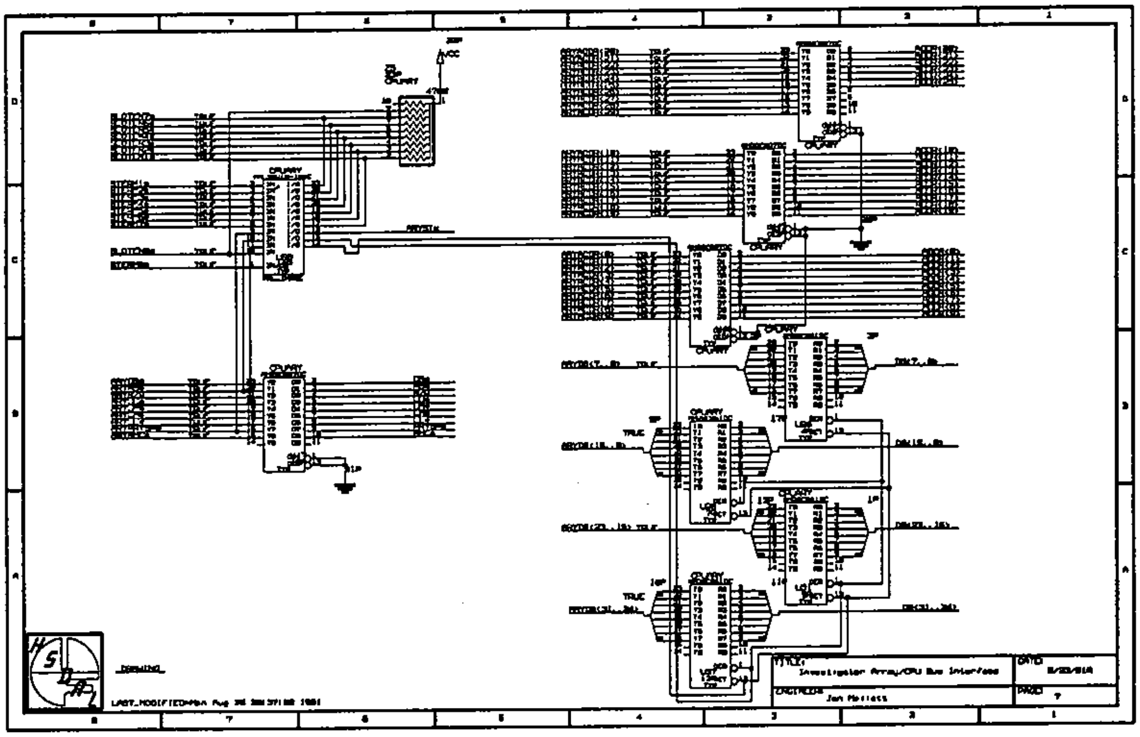


Figure A.7: InvestiGATOR Array Bus Interface

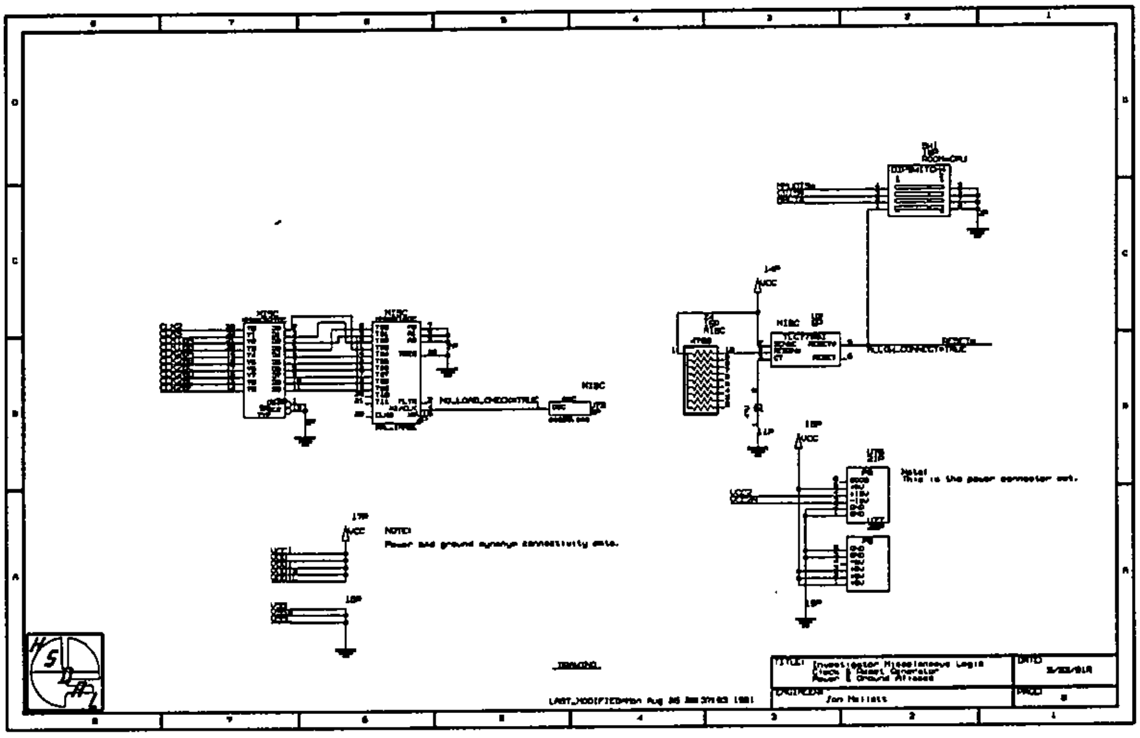


Figure A.8: InvestiGATOR MIscellaneous Module

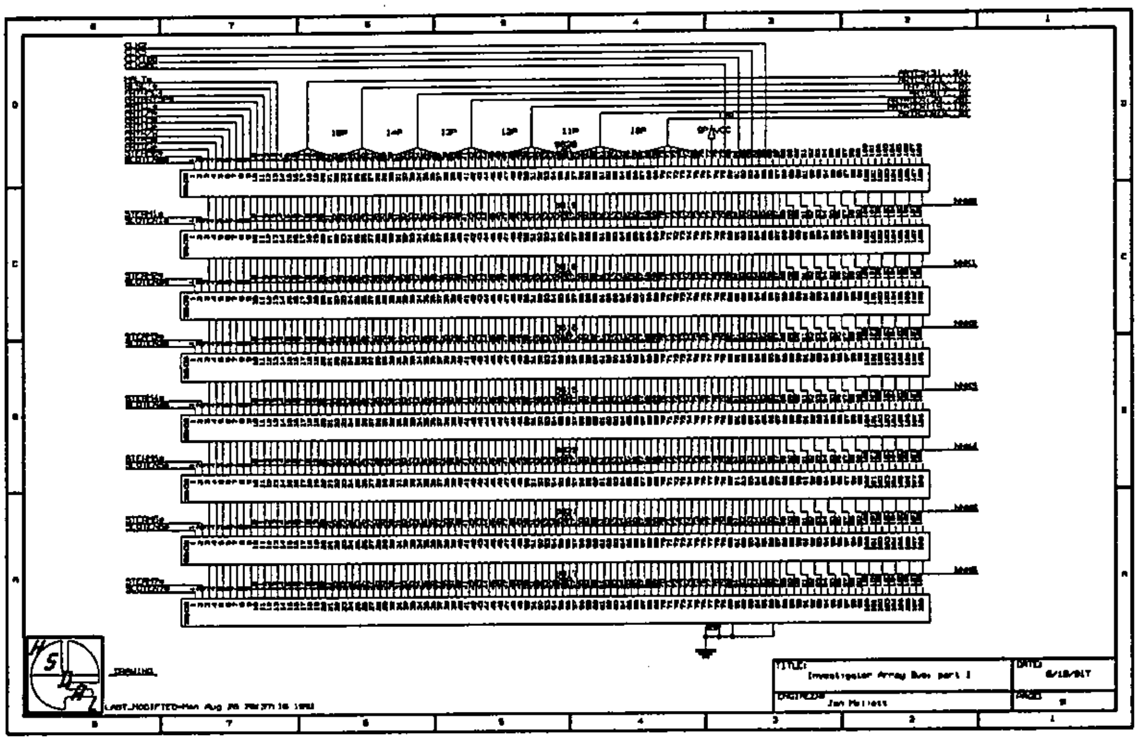


Figure A.9: InvestiGATOR Array Bus, First Part

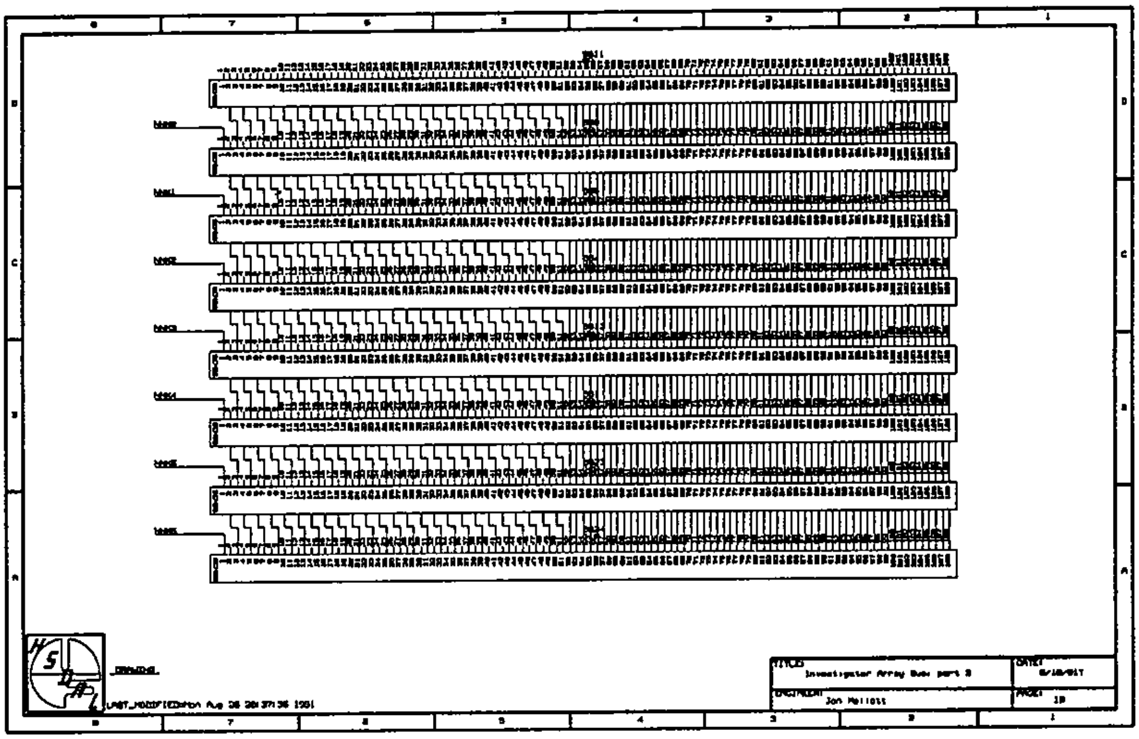


Figure A.10: InvestiGATOR Array Bus, Second Part

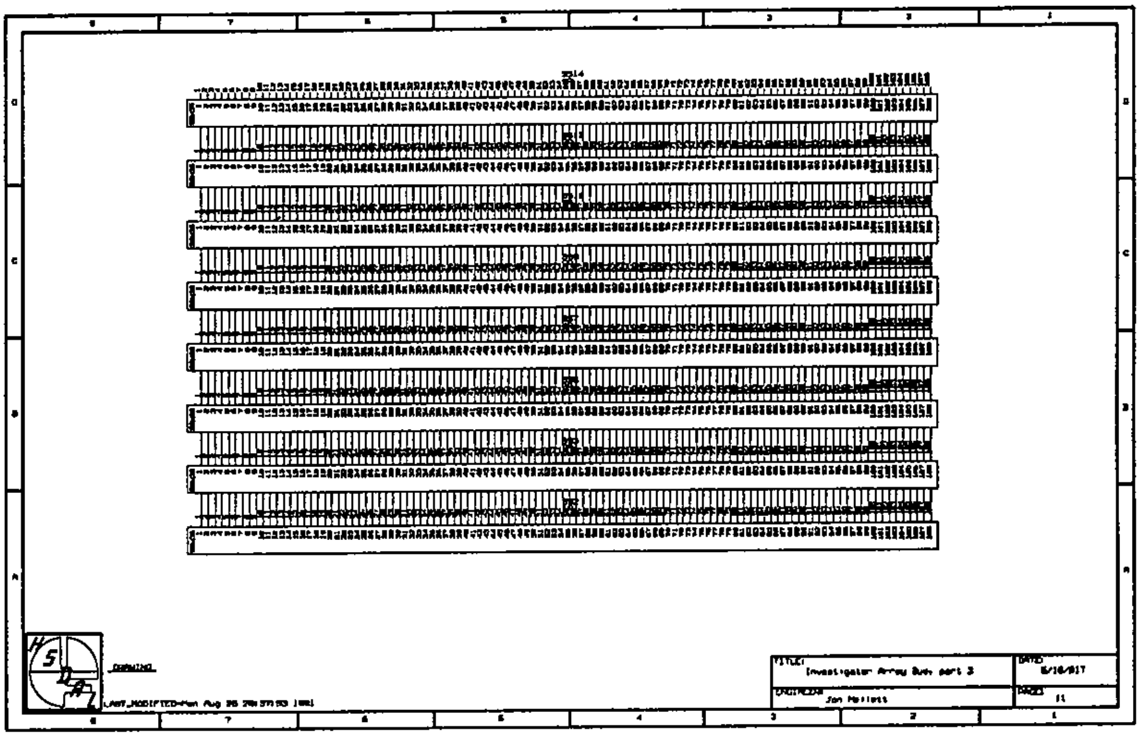


Figure A.11: InvestiGATOR Array Bus, Third Part

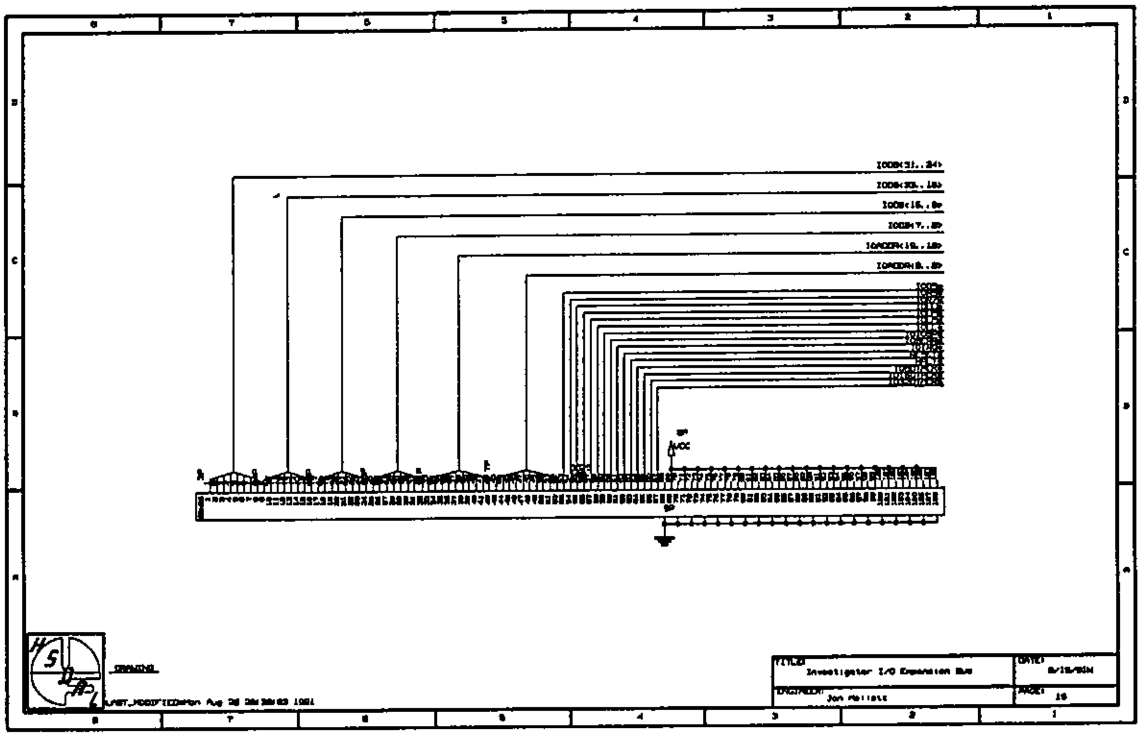


Figure A.12: InvestiGATOR I/O Expansion Bus

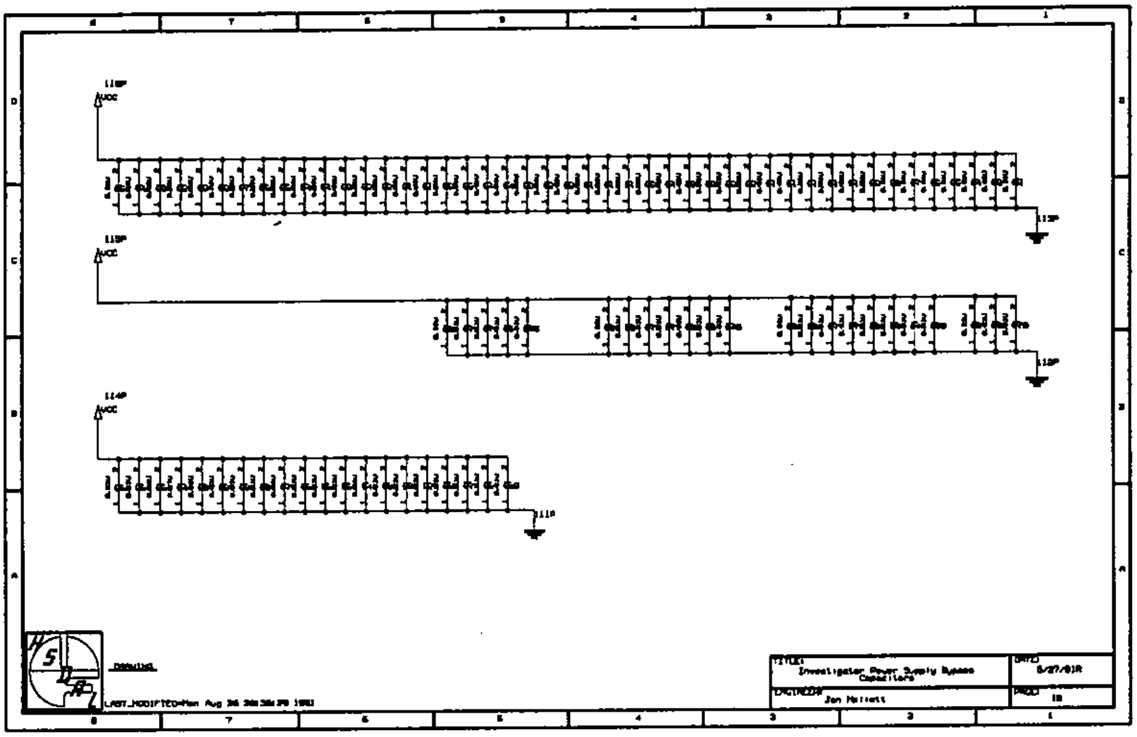


Figure A.13: InvestiGATOR Bypass Capacitors

# Appendix B

# INVESTIGATOR STATE MACHINES

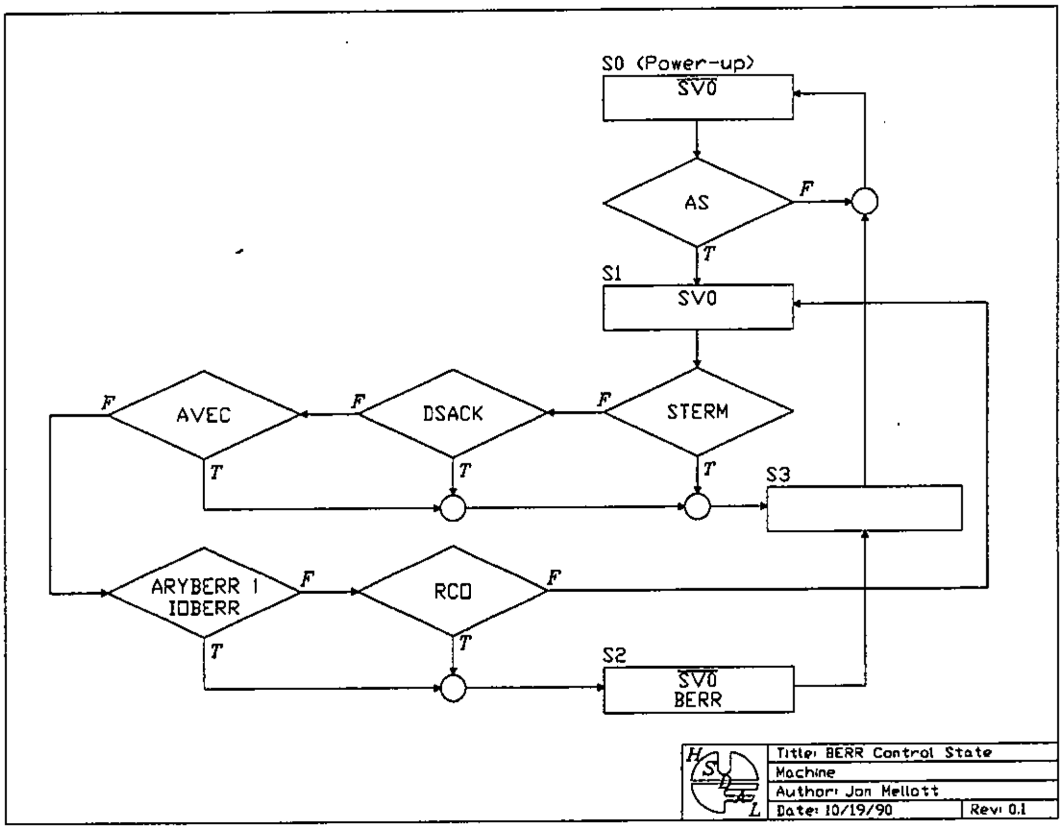


Figure B.1: Bus Error Detection State Machine

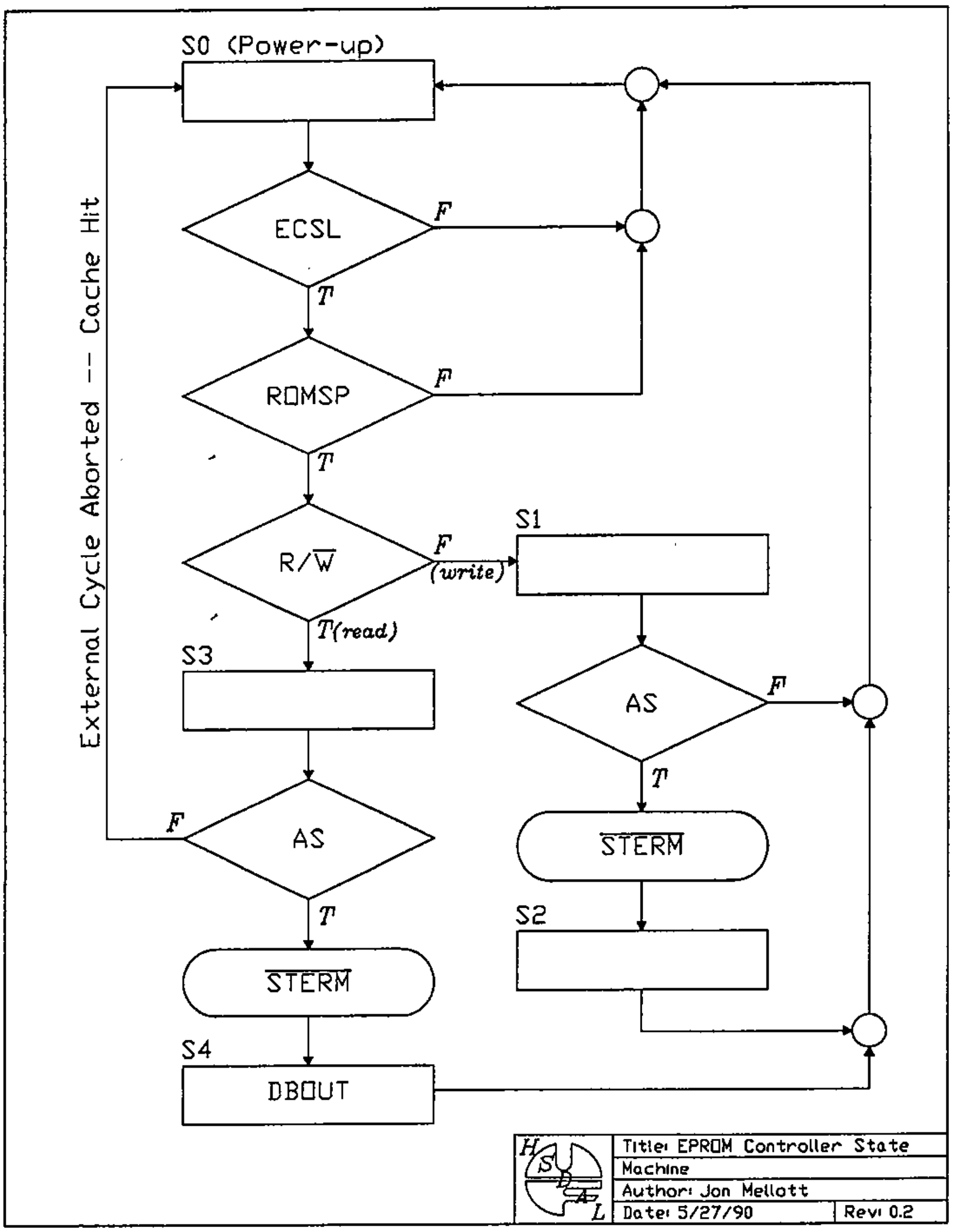


Figure B.2: ROM Controller State Machine

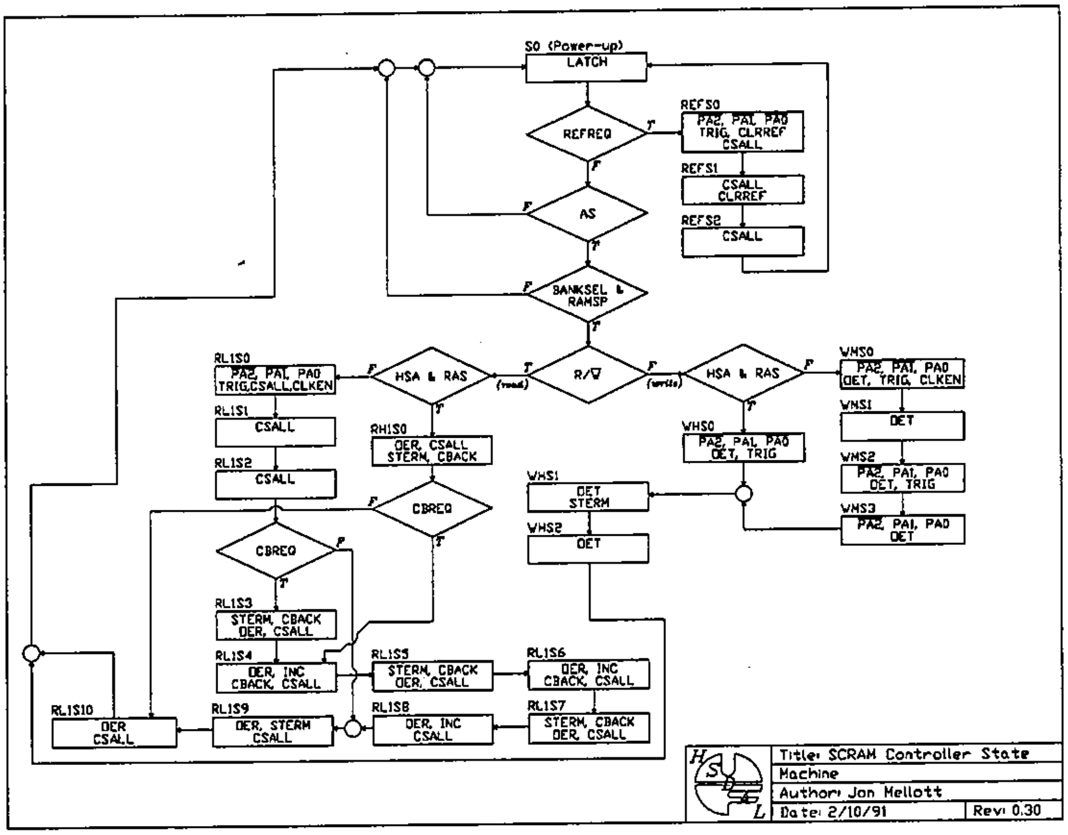


Figure B.3: SCRAM Controller State Machine

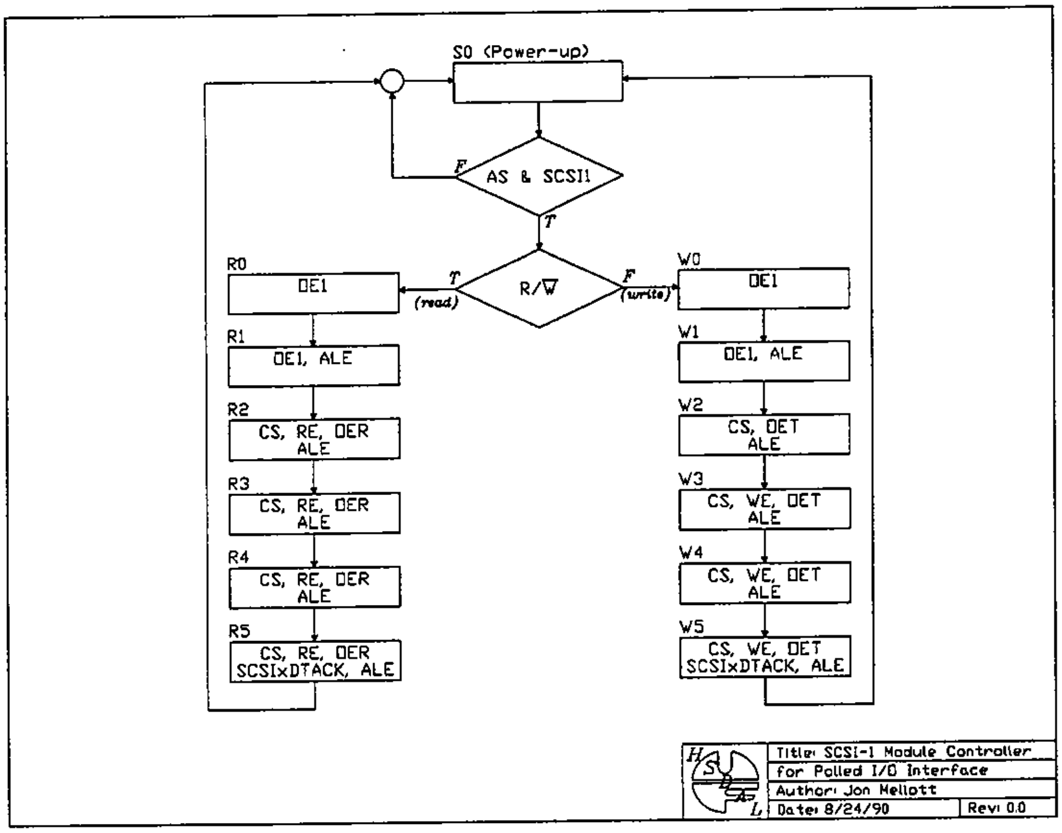


Figure B.4: SBIC Controller State Machine

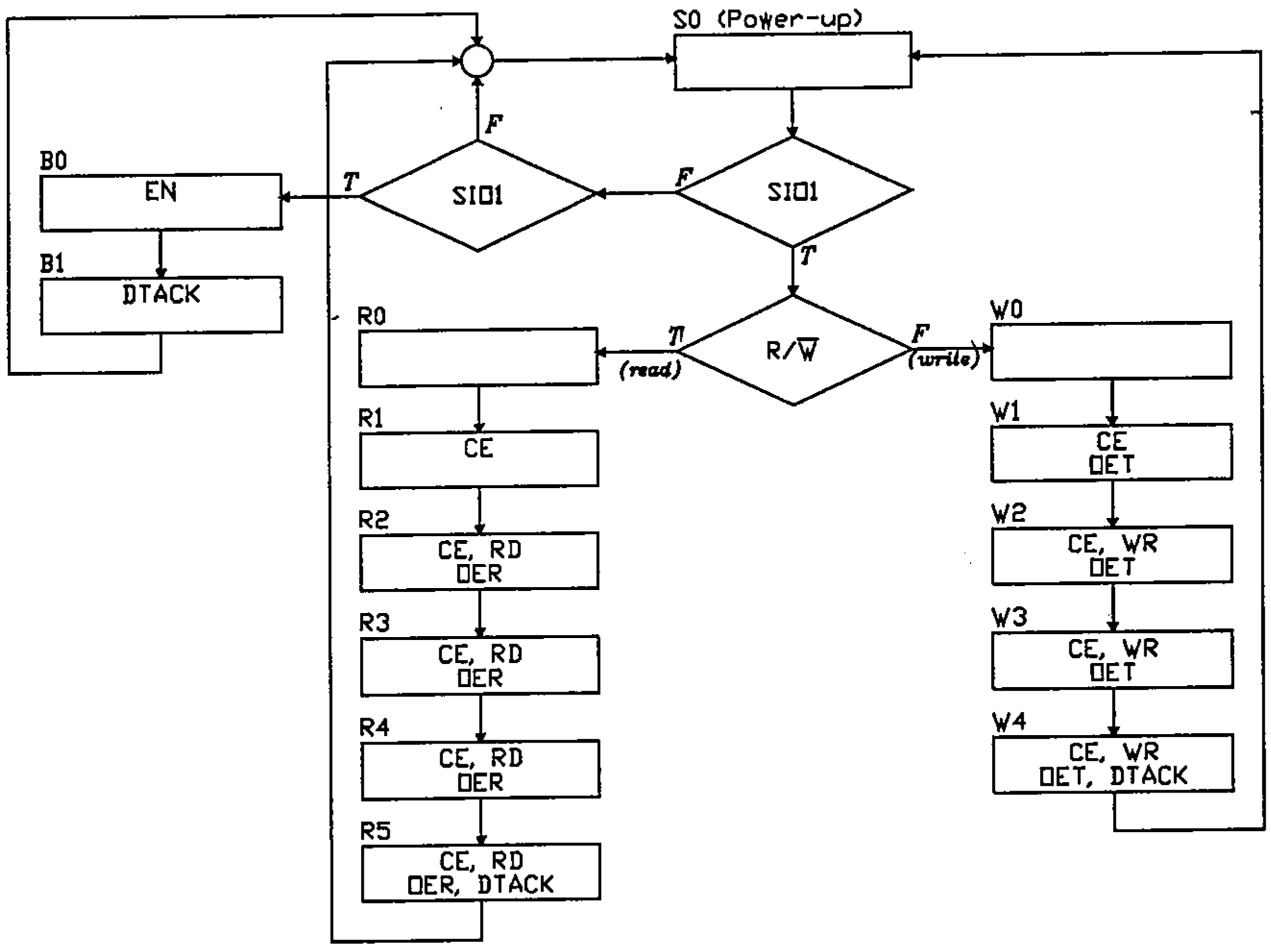


Figure B.5: SIO Controller State Machine

Appendix C

# INVESTIGATOR PROGRAMMABLE LOGIC DEVICE LISTINGS

This appendix contains lists for the InvestiGATOR PLDs. These PLDs are used throughout the InvestiGATOR.

NOTE: THIS MATERIAL IS AVAILABLE UPON REQUEST.

## C.1 MACH1C

## C.2 MACH2

## C.3 MACH3XA

## C.4 PAL0

## C.5 PAL1A

## C.6 PAL3B

## C.7 PAL4

## C.8 PAL5

## C.9 PAL7

## C.10 PAL12

## Appendix D

## INVESTIGATOR SOURCE CODE

NOTE: THIS MATERIAL IS AVAILABLE UPON REQUEST.

# Appendix E

# GAUSS MACHINE SCHEMATICS

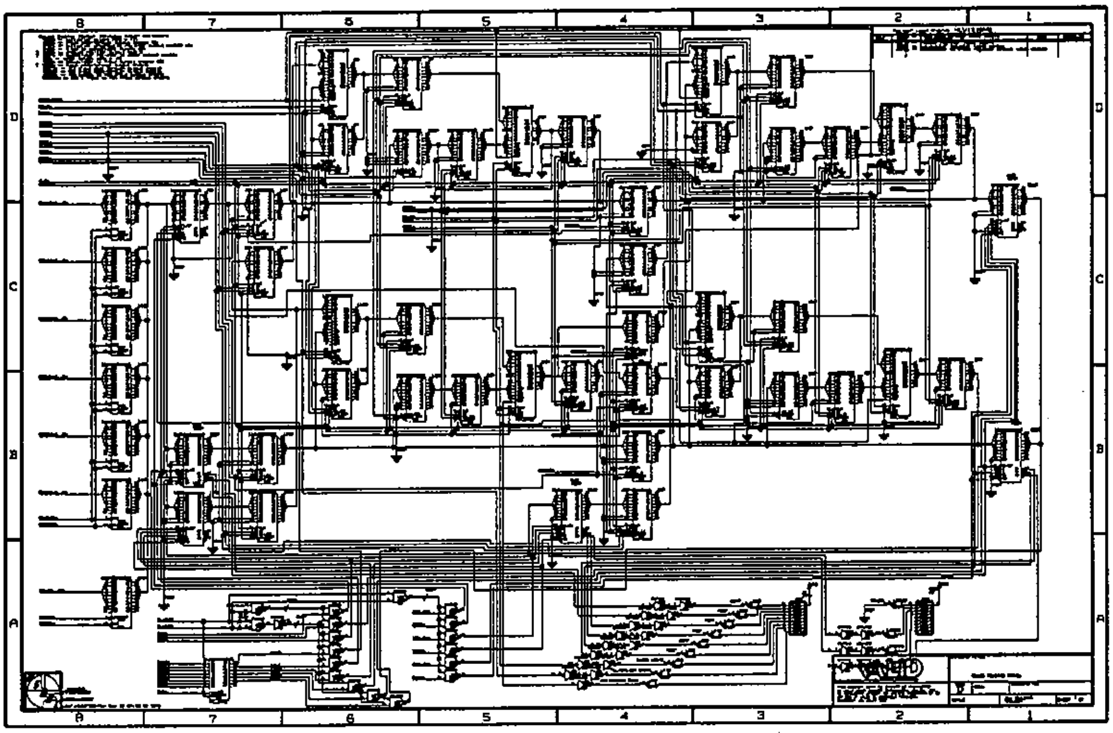


Figure E.1: Gauss Array

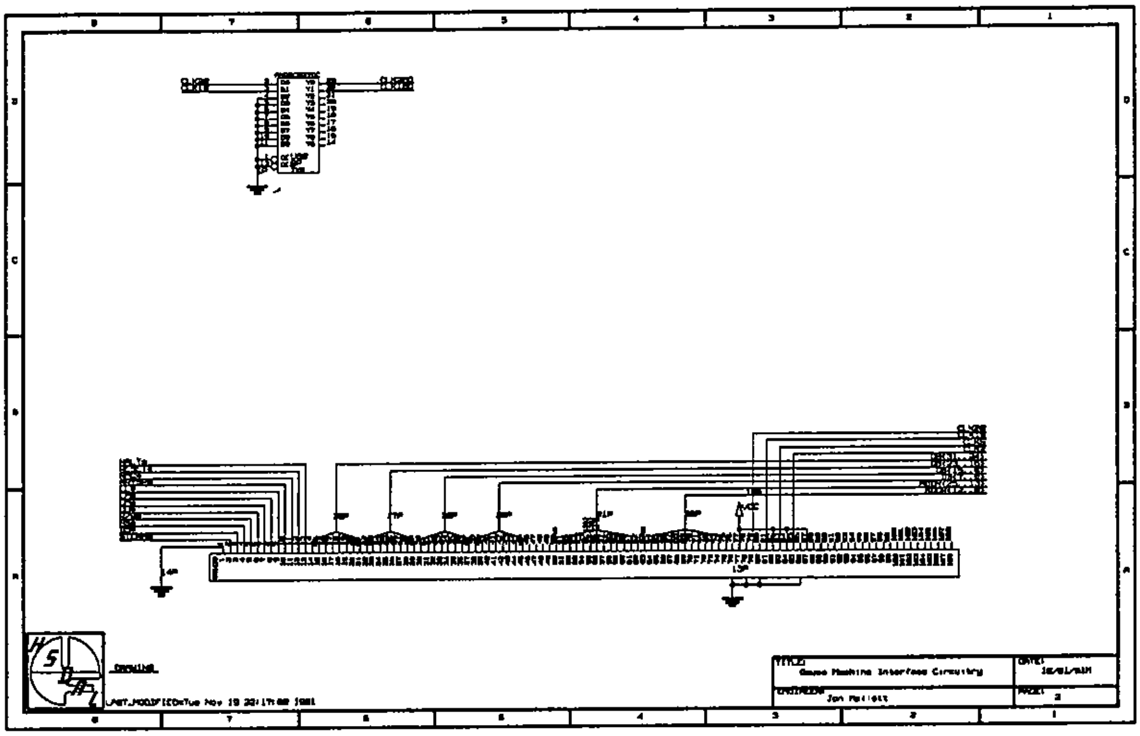


Figure E.2: Gauss Array Miscellaneous

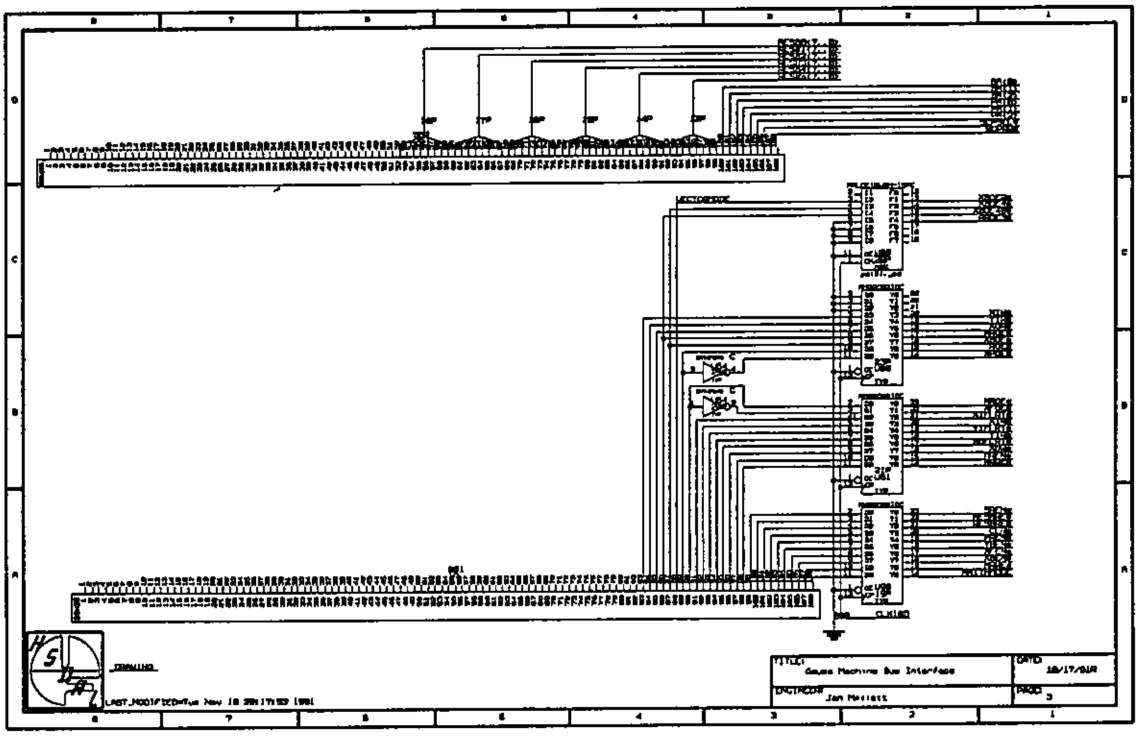


Figure E.3: Gauss Array Instruction Decoding

Appendix F

# GAUSS MACHINE PROGRAMMABLE LOGIC DEVICE LISTINGS

This appendix contains listings for the Gauss machine controller's PLDs. These PLDs are used for decoding instructions from the microsequencer, and interface glue between the InvestiGATOR and the controller.

NOTE: THIS MATERIAL IS AVAILABLE UPON REQUEST.

## F.1 PAL Listings

### F.1.1 PALC1.PDS

### F.1.2 PALC2.PDS

### F.1.3 PALC3.PDS

### F.1.4 PALC4.PDS

### F.1.5 PALC5.PDS

### F.1.6 PALC6.PDS

### F.1.7 PALC7.PDS

### F.1.8 PALC8.PDS

### F.1.9 PALC9.PDS

# Appendix G

# GAUSS MACHINE MICROCODE

This appendix lists the Gauss machine microcode. This microcode is written for Advanced Microdevices' ASM14x microcode assembler. This microcode assembler emits object code for AMD's Am29PL14x/15x family of EPROM based microsequencers. The listing in Section G.1 is the source for the microcode. Section G.2 describes the microcode and microinstructions.

## G.1 Gauss Machine Microcode Listing

NOTE: THIS MATERIAL IS AVAILABLE UPON REQUEST.

## G.2 Gauss Machine Microcode Description

Appendix H

## MACINTOSH API SOURCE CODE

This appendix contains source code for the Macintosh/Gauss machine API.

NOTE: THIS MATERIAL IS AVAILABLE UPON REQUEST.

### H.1 TYPES.H

### H.2 CONV.H

### H.3 INTMATRIX.H

### H.4 INTMATRIX.C

### H.5 LIST.H

### H.6 LIST.C

### H.7 MATRIX.H

### H.8 MATRIX.C

### H.9 UTILS.H

### H.10 UTILS.C

## REFERENCES

[1] M. Griffin, F. J. Taylor, and M. Sousa, "New scaling algorithms for the chinese remainder theorem," in *Proc. 22nd Asilomar Conf. on Signals, Syst., and Computers*, 1988.

[2] M. Griffin, M. Sousa, and F. J. Taylor, "Efficient scaling in the residue number system," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, 1989.

[3] J. V. Krogmeier and W. K. Jenkins, "Error detection and correction in quadratic residue number systems," in *26th Midwest Symposium on Circuits and Systems*, 1983.

[4] W. K. Jenkins, "The design of error checker for self-checking residue number arithmetic," *IEEE Trans. Computers*, vol. 32, pp. 388–396, Apr. 1983.

[5] G. Zelniker and F. J. Taylor, "A reduced complexity finite field alu," *IEEE Trans. on Circuits and Systems*, vol. 38, pp. 1571–1573, Dec. 1991.

[6] A. van de Goor and C. Verruijt, "An overview of deterministic functional ram chip testing," *ACM Computing Surveys*, vol. 22, pp. 5–34, March 1990.

[7] S. Y. Kung, "VLSI array processors," *IEEE ASSP Magazine*, pp. 4–22, July 1985.

[8] E. Arnould, H. T. Kung, O. Menzilcioglu, and K. Sarocky, "A systolic array computer," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, 1985.

[9] H. T. Kung and others;, "iWarp: an integrated solution to high-speed parallel computing," *IEEE Trans. Computers*, vol. 38, pp. 330–339, Sep. 1988.

[10] R. Simar, "The TMS320C40: a DSP for parallel processing," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, pp. 1089–1092, 1991.

[11] G. H. Golub and C. F. van Loan, *Matrix Computations.* Baltimore: Johns Hopkins University Press, 2nd ed., 1989.